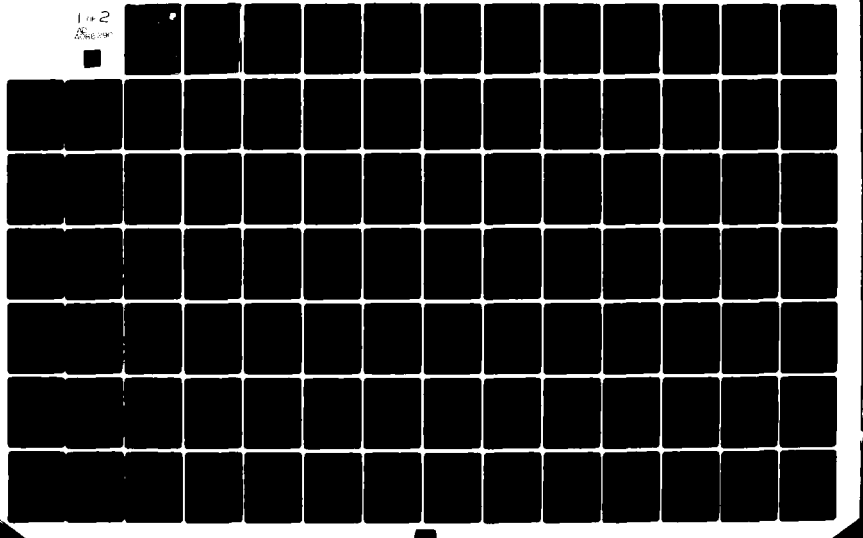
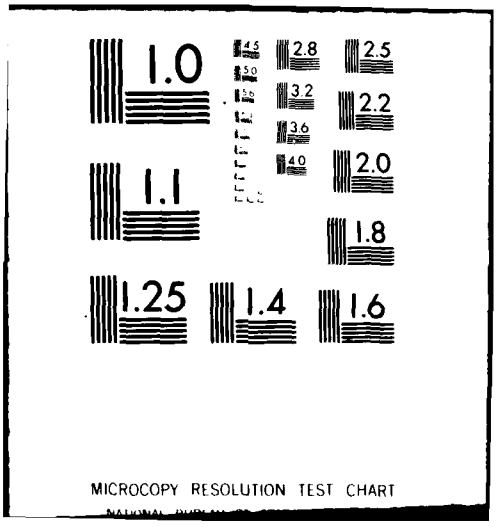
AD-A086 290 NORTHWESTERN UNIV EVANSTON IL DEPT OF ELECTRICAL ENG--ETC F/G 9/2
SELF-METRIC SOFTWARE. VOLUME I. SUMMARY OF TECHNICAL PROGRESS.(U)
APR 80 S S YAU F30602-76-C-0397
UNCLASSIFIED RADC-TR-80-138-VOL-1 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART

LEVEL II

RADC-TR-80-136 Vol I (of three)
Final Technical Report
April 1980

SELF-METRIC SOFTWARE
Summary Of Technical Progress

Northwestern University

Stephen S. Yau

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-80-138, Vol I (of three) has been reviewed and is approved for publication.

APPROVED: Rocco F Iuorno

ROCCO F. IUORNO
Project Engineer

APPROVED: Wendall C Bauman

WENDALL C. BAUMAN, Col, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
(18) 1. REPORT NUMBER: RADC-TR-80-138, Vol-1 (of three)	2. GOVT ACCESSION NO.: AD-A086290 (7) — 3. RECIPIENT'S CATALOG NUMBER
(6) 4. TITLE (and Subtitle): SELF-METRIC SOFTWARE • Volume I • Summary of Technical Progress,	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report, 1 Aug 76 — 15 Jan 80
	6. PERFORMING ORG. REPORT NUMBER: N/A
(10) 7. AUTHOR(s): Stephen S. Yau	(15) 8. CONTRACT OR GRANT NUMBER(s): F30602-76-C-0397
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Northwestern University, Department of Electrical Engineering & Computer Science, Evanston IL 60201	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55810278 (16) (17) 20
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441	(11) 12. REPORT DATE: April 1980
	13. NUMBER OF PAGES: 110
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same (12) 110	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Rocco F. Iuorno (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software maintenance, self-metric software, logical and performance ripple effect analysis, testing during software maintenance, specification for program modifications, dynamic monitoring of software behavior, quality factors for software maintainability, stability and understandability.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report documents the research performed under RADC Contract ~~F30602-76-C-0397~~ by Northwestern University in the area of developing effective techniques for large-scale software maintenance, including those for the design, implementation, validation, and evaluation of reliable and maintainable software systems with a high degree of automation. During this contract period, research in the areas of ripple effect analysis, testing during software maintenance, specification for program modifications,

over

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED (Cont'd)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Item 20 (Cont'd)

quality factors for software maintainability, and dynamic monitoring of program behavior was conducted. In this report, the software maintenance process is first described. The research results which have been presented in previous papers and interim technical reports are summarized, and unfinished work is presented.

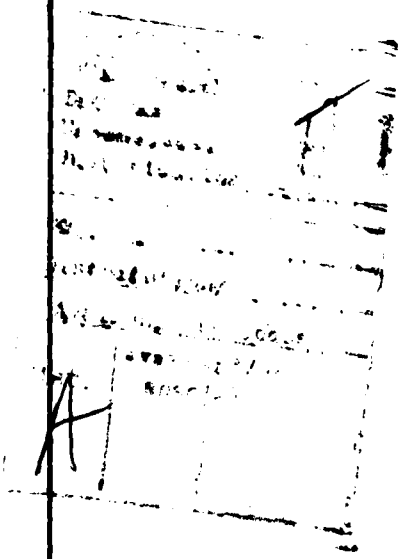


TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Evaluation:

The purpose of this research was to study and develop techniques to make the software maintenance process more efficient The effort was initiated in response to requirements defined in the RADC Technical Plan in Software Engineering, TPO 4G3. This subthrust supports the development of software concepts and tools required for the Software development and software maintenance of Air Force systems.

The research, which covered three years of effort produced four interim reports, a final report in three volumes and numerous papers. Topics investigated included ripple effect analysis, software testing, specification for program modification, and quality factors for software maintenance.

The technical discussion provided in the interim reports and in Vol I of the final report can be used to further the development of software maintenance tools to enhance the maintenance process. The technical discussion on ripple effect analysis supports the Handbook generated. The discussion on effective testing provides information on how to test software after changes are made. The material on the generation of specifications for proposed modifications makes use of recent work in formal specification languages in an attempt to provide clear, unambiguous change orders to software personnel. The material on

stability measures and dynamic monitoring can be used by those who are attempting to establish criteria for assessing the quality of software. Volume II and III of the final report represent a Handbook which can be used effectively by software maintenance personnel actively engaged in making software changes. The Handbook can guide them on determining what sections of code will be impacted by a modification and how the modification can affect the logical and performance characteristics of the original software.

The fruitful results obtained, which represent a good start on defining and controlling the software maintenance process, have warranted the initiation of a follow-on effort to continue studying the problem and to refine the concepts and techniques developed.

Rocco F. Iuorno

ROCCO F. IUORNO
Project Engineer

1.0 OBJECTIVE

This report summarizes the research performed under Contract No. F30602-76-C-0397 by Northwestern University for Rome Air Development Center during the period of August 1, 1976 to January 15, 1980. Research results which have been presented in previous papers and interim technical reports are summarized, and unfinished work is presented. Publications, presentations and technical personnel related to this project are described.

The objective of this effort is to conduct applied research for the development of effective techniques for the design, implementation, validation, and evaluation of reliable and maintainable software systems with a high degree of automation. The following areas of research are to be emphasized: methodologies for the design and implementation of easily maintainable software; techniques for static and dynamic analysis and measurement of computer programs; techniques for evaluating and improving the maintainability of existing computer programs.

During this contract period, significant results have been obtained in developing techniques which contribute to effective maintenance of large-scale software, and in evaluating the maintainability of computer programs. Before discussing our research results, let us first examine the maintenance process and quality factors affecting software maintainability.

2.0 SOFTWARE MAINTENANCE PROCESS AND ASSOCIATED QUALITY FACTORS

It is noted that software maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, optimization, and minor changes in mission requirements [1-7]. Once a particular objective for the maintenance activity is established, the objective can be accomplished in four phases as described in Figure 1 [8].

The first phase consists of analyzing the program in order to understand it. Several factors such as the complexity of the program, the documentation, and the self-descriptiveness of the program contribute to the ease of understanding the program.

The second phase consists of generating a particular maintenance proposal to accomplish the implementation of the maintenance objective. This requires a clear understanding of both the maintenance objective and the program to be modified. Several factors, such as the extensibility of the program,

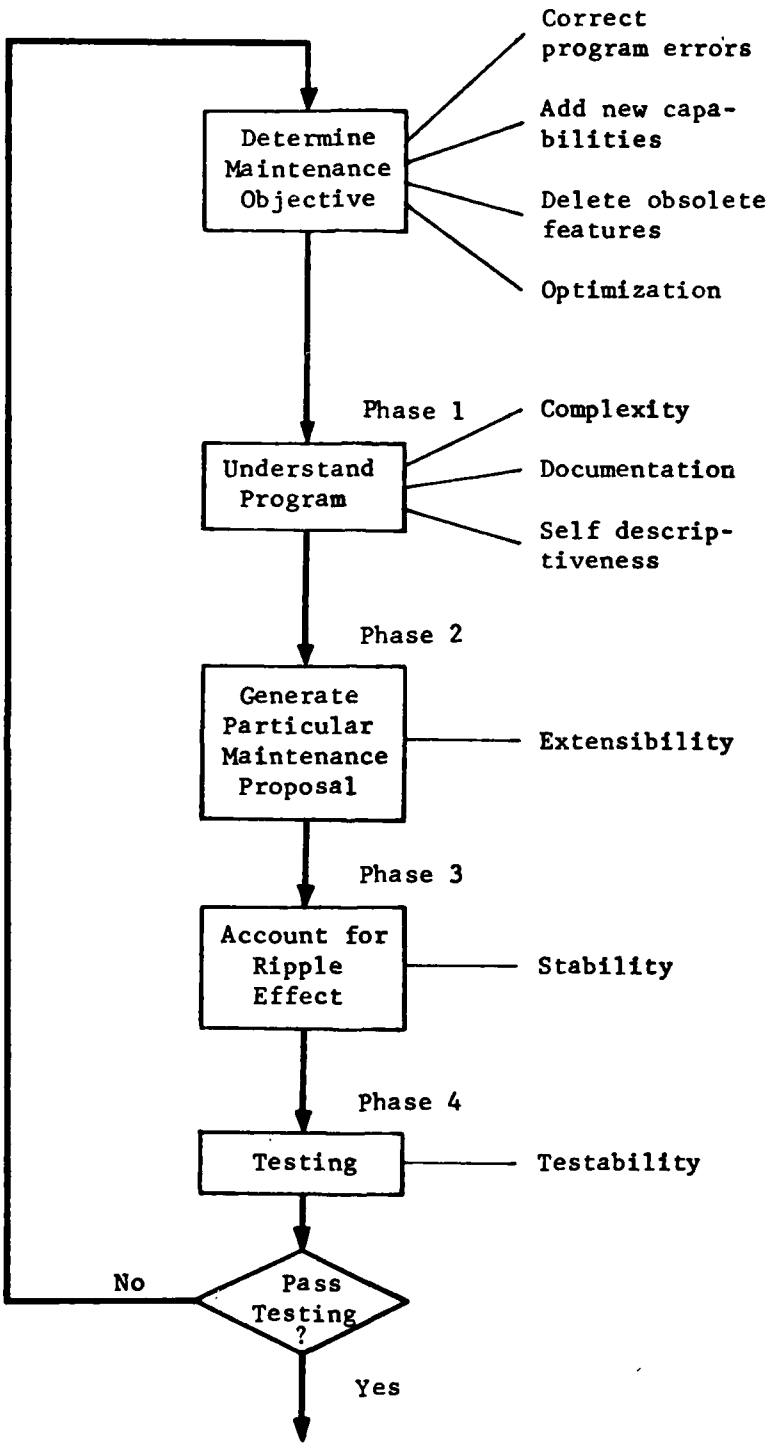


Figure 1. The Software Maintenance Process

contribute to the generation of various program maintenance proposals with ease.

The third phase consists of accounting for all of the ripple effect as a consequence of the modification. In software, the effect of a modification may not be local to the modification, but may also affect other portions of the program. There is a ripple effect from the location of the modification to the other parts of the program that are affected by the modification. One aspect of this ripple effect is logical or functional in nature. Another aspect of this ripple effect concerns the performance of the program. During software maintenance, it is possible to perform a modification to the program, investigate its logical ripple effect and locate the inconsistencies introduced throughout the program by the modification. After all the logical corrections have been made to the program, the maintenance personnel may conclude that they have restored the program to its previous level of functional correctness. The performance of the program, however, may have been altered as a direct result of this maintenance activity. Since a large-scale program usually has both functional and performance requirements, the net result of the maintenance effort may be satisfactory to the functional requirements, but not to some performance requirement. In many large-scale programs, the violation of a performance requirement is equivalent to a program error and, thus, requires further corrective action. Consequently, it is important in the maintenance process to fully understand the potential effect of a modification to the program from both a logical and performance point of view. The primary factor affecting the ripple effect as a consequence of a program modification is the stability of the program. Program stability is defined as the resistance to the amplification of program modifications.

The fourth phase consists of testing the modified program to insure that it is correct. It is important that cost-effective testing techniques be applied during maintenance. The primary factor contributing to the success of these cost-effective techniques is the testability of the program. Program testability is defined as a measure of how little effort is required to test the program. Each of these phases for accomplishing the maintenance objective along with the major software quality factors affecting each phase of the maintenance process is described in Figure 1.

With the above discussion, it is easily seen where our results contribute to large-scale software maintenance. Our results can be summarized in the following areas: 1. ripple effect analysis, 2. generation of specification for program modifications, 3. effective testing for software maintenance, 4. quality factors for software maintainability, and 5. dynamic monitoring of software behavior. We will summarize our results in each of these areas in following sections.

3.0 RIPPLE EFFECT ANALYSIS

As discussed before, a significant factor contributing to the complexity and cost of performing software maintenance is the fact that the effect of a program modification may not be local to the location of modification, but may also affect other portions of the program. This constitutes a ripple effect from the location of modification to other parts of the program that are affected by the modification. The ripple effect has both a functional and performance perspective [1].

We have developed a ripple effect analysis methodology for both logical and performance ripple effect analysis. The techniques can be developed as a powerful tool to help maintenance personnel understand the scope of effect of their changes on the program by identifying the parts of the program which must be checked for logical consistency and sources for performance degradation.

A series of technical reports and papers has been prepared to describe our results in this area [1,9-11]. A handbook [9] has been prepared to describe our ripple effect analysis methodology. The detailed description as well as the capabilities and limitation of the logical ripple effect analysis technique is contained in Section 3.1. The detailed description of the performance ripple effect analysis technique is contained in the technical reports [10,11]. In this section, logical ripple effect will be analyzed through the development of an intramodule error flow model and an intermodule error flow model. A logical ripple effect analysis technique based upon these models will also be developed. Some implementation considerations for developing a tool to perform logical ripple effect analysis on programs written in a subset of JOCIT JOVIAL language will also be discussed.

3.1 Logical Ripple Effect Analysis Models

In this section, models will be developed for the analysis of logical ripple effect. The analysis of logical ripple effect corresponds to an analysis of how logical errors may propagate through the program. This error flow can be simplified by decomposing it into the error flow which occurs within a module and the error flow which occurs between modules. An understanding of how error flow occurs both within a module and among modules is sufficient to understand how error flow occurs in the entire program. Thus, in this section, the error flow in the program will be modelled utilizing an intramodule error flow model and an intermodule error flow model.

3.1.1 Intramodule Error Flow Model

In order to develop a model of how the error flow occurs within a module, it is first necessary to develop a model of a program module. A program module is defined to be a separately invokable piece of code having single entry and single exit points. Practically speaking, a module can correspond to a SUBROUTINE or PROCEDURE, etc. To reduce complexity, our program module model represents a module as a set of program blocks. A program block is a maximal sequence of computer statements having the property that each time any statement in the sequence is executed, all are executed; except when a module invocation is encountered. Each program block has a single entry point and a single exit point. The flow of control among these program blocks is then represented in our program module model by a digraph G, which consists of a set of vertices, $V = \{v_1, v_2, \ldots, v_i, \ldots, v_k\}$, representing the set of program blocks and a set of branches, B, of ordered pairs of vertices representing the flow of control from the exit point of a program block to the entry of another program block. The set of blocks which are immediate predecessors of a block v_i is denoted by $I^{-1}(v_i)$, and the set of blocks which are immediate successors of v_i is denoted by $I(v_i)$.

The program module model can be utilized to simplify intramodule error flow by decomposing it into the error flow which occurs within a program block and the error flow which occurs between program blocks in the module. In order to analyze the error flow within program blocks and between program blocks, it is necessary to develop a characterization for a program block which reflects how potential errors flow within the block.

The basis for the characterization of a program block requires the identification of all data items, control items, data definitions, control definitions, data usages, and control usages in the program block. A data item is a member of the set of minimal information units which describe the program. They basically consist of the program's variables. A control item is assigned for each control directive which determines the execution of a statement or a sequence of statements. The predicate in a conditional statement provides a control directive which determines the outcome of the decision point, and hence a control item is assigned to represent the predicate. An iterative statement which establishes a controlled loop also provides one type of control item. A control item can be artificially defined in a manner that it will not create any error flow in the program by assigning to it a symbolic name which is guaranteed to be unique in the program. This can easily be accomplished by specifying a format for control items which is absolutely different from that of any name in any programming language.

A data definition is a data item whose value is modified in an expression or part of an expression. A data usage is a data item which is referenced without change in an expression or part of an expression. A control definition is a control item whose associated control directive is defined in an expression or part of an expression. A control usage is a control item whose associated control directive may affect a data or control definition in an expression or part of an expression. For example, in a DO statement, (DO 100 I = A,B,C), the control item assigned for the control directive is a control definition while program variables A, B and C are data usages identified from the DO statement.

It is assumed in our program block characterization that all data items have a unique memory address and that this memory address can be symbolically determined prior to program execution. A data item is said to employ explicit addressing if the memory address of the data item can be symbolically determined prior to program execution; otherwise, it is said to employ implicit addressing. A control item is considered as employing explicit addressing although there is no memory address for it. An example of implicit addressing of data items is the array data structure. The integrity of a data structure employing implicit addressing can only be kept if no element in the data

structure is affected by the error flow.

Tracing the exact error flow for programs which contain data structures employing implicit addressing is infeasible due to the inability to precisely define the error behavior of these data structures. However, the worst error flow can be computed by treating a data structure as a single data item. Thus, if an element in a data structure is affected by the error flow, the whole data structure is considered as being affected by the error flow.

Utilizing this basis, the error flow characterization of a program block v_i can be described by the sets P_i and C_i and the mapping $\bar{f}_i$ all of which are defined by the following definitions:

Definition 1. The source capable set C_i of a program block v_i is the set of all definitions within v_i, in which each element can cause potential errors to exist in v_i.

C_i can be partitioned into two subsets: C_{r_i} which contains the definitions employing explicit addressing, and $\bar{C}_{r_i}$ which contains the definitions employing implicit addressing.

Definition 2. The potential propagator set P_i of a program block v_i is the set of usages within v_i which can propagate potential errors from outside of v_i to some elements in C_i. A usage x in v_i is an element in P_i if it is used before or without any redefinition to it.

Definition 3. The flow mapping $\bar{f}_i: P_i \rightarrow C_i$ is a mapping which maps each element p in P_i to a set of elements in C_i such that p can propagate potential errors to these elements in C_i.

The set of elements in C_i which is mapped by an element p in P_i under $\bar{f}_i$ is denoted by $\bar{f}_i(p)$. By definition,

$$\bar{f}_i(p) = \{c \in C_i \mid p \text{ can propagate potential errors to } c\}. \quad (1)$$

The set of elements in C_i which is mapped by some elements in P_i under $\bar{f}$ is denoted by $\bar{f}_i(P_i)$. Obviously,

$$\bar{f}_i(P_i) = \bigcup_{p \in P_i} \bar{f}_i(p). \quad (2)$$

The determination of this error flow characterization for each program block in the module enables the error flow within program blocks and between program blocks in the module to be modeled by our intramodule error flow

model. This intramodule error flow model describes the error flow within the module in terms of primary, secondary, and propagation error source sets which are defined as follows:

Definition 4. A primary error source is a definition which is involved in the initial modification. The primary error sources identified in a program block v_i constitute the primary error source set (LK_i) of v_i.

Definition 5. A definition x in v_i is implicated as a secondary error source by an element p in P_i if $x \in \bar{f}_i(p)$ and p is a primary or secondary error source itself.

Various primary and secondary error sources together form propagation error source sets for various blocks. The propagation error source set of a program block v_i is formally defined as follows:

Definition 6. The propagation error source set δ_i is the set of error sources which flow out of v_i. The module propagation error source set δ is defined as $\bigcup_i \delta_i$, where i indexes the blocks in the module.

The error flow between program blocks is modeled by calculating the set of error sources which flow out of a program block v_i given a propagation error source set δ_j which flows into v_i and a primary error source set LK_i for v_i. An error source e flows out of v_i if it satisfies at least one of the following conditions:

1) e is a primary error source identified in v_i,
2) e is implicated in v_i as a secondary error source, or
3) e is an incoming error source which passes through v_i.

Thus, the set of all error sources which flow from v_i can be calculated as the union of the sets of error sources each of which contains all of the error sources satisfying one of the above conditions. Each of these sets can easily be calculated. The set of error sources satisfying condition 1 is the primary error source set LK_i of v_i. As for condition 2, an element x in $P_i \cap \delta_j$ is capable of implicating a set of secondary error sources because x is an incoming error source and it can propagate potential errors to some definitions in v_i. The set of secondary error sources implicated by x is provided by the flow mapping on x, i.e. $\bar{f}_i(x)$ is the desired set. Thus, $\bar{f}_i(P_i \cap \delta_j)$ is the set of all secondary error sources implicated in v_i by an incoming error source set δ_j. As for condition 3, an incoming error source x cannot

pass through v_i if it employs explicit addressing and it is redefined in v_i; in other words, x cannot pass through v_i if it is an element in C_{r_i}. Hence, the set of incoming error sources which pass through v_i is given by $\delta_j - (\delta_j \cap C_{r_i})$.

The error flow between program blocks can thus be modeled utilizing a tracing function for calculating the set of error sources which flows from v_i given an error source set δ_j which flows into v_i and a primary error source set LK_i for v_i. The tracing function, f, can be defined by

$$f(v_i,\delta_j) = \{[\delta_j - (\delta_j \cap C_{r_i})] \cup \bar{f}_i(\delta_j \cap P_i) \cup LK_i\}. \quad (3)$$

The error flow from the point of initial modification to other program areas can then be modelled utilizing this tracing function along with the error characteristics of the program's blocks and the program graph. The pair (v_i,δ_i) can be associated with each block v_i in the module, where δ_i is the set of error sources which flow from v_i. To trace the error flow from a block s involved in the initial modification, δ_s can be initialized by LK_s. The program graph representing the paths between program blocks must then be examined. If there is only one path between the primary error source block s and some arbitrary block v_i, then δ_i can be calculated by the successive applications of the following tracing function,

$$\delta_i = \{f(v_i,f(v_{i-1},\ldots,f(v_1,\delta_s))\ldots) \mid s,v_1,\ldots v_i \text{ is the valid path from s to } v_i\}. \quad (4)$$

If multiple paths exist between s and v_i, then

$$\delta_i = \bigcup_p \{F_i(p) \mid p \text{ is a valid path from s to } v_i\}, \quad (5)$$

where

$$F_i(p) = \{f(v_i,f(v_{i-1})\ldots,f(v_1,\delta_s))\ldots) \mid p=s,v_1,\ldots,v_i\}. \quad (6)$$

Furthermore, if the set of blocks $S = \{s_1,s_2,\ldots,s_k,\ldots,s_n\}$ within the module has the initial modification and if δ_k is initialized by LK_k for each $s_k \in S$, then

$$\delta_i = \bigcup_{s_k \in S} \{\bigcup_p F_i(p) \mid p \text{ is a valid path from } s_k \text{ to } v_i\}, \quad (7)$$

where

$$F_i(p) = \{f(v_i,f(v_{i-1},\ldots,f(v_1,\delta_k))\ldots) \mid p=s_k,v_1,\ldots,v_i\}. \quad (8)$$

These tracing functions provide the ability to compute all of the propagation error source sets for all of the blocks in the module affected by the modification. Our intramodule error flow model refines this data to distinguish between blocks which must be examined to insure consistency with the modification and those which need not be examined. This is accomplished by utilizing a block identification criterion which states that a block requires no further maintenance if no error source which flows from the block is internally generated in the block.

After the propagation error source sets of all the blocks in the module have been computed, the block identification criterion put forth in the following lemma is applied to each block to determine whether the block requires further maintenance effort.

Lemma 1. (Block Identification Criterion). If for a program block v_i, $\mathcal{S}_i \cap C_i = \emptyset$, then v_i requires no further maintenance effort.

Proof. $(\mathcal{S}_i \cap C_i) = \emptyset$ implies that no error source is internally generated in v_i. Thus, the consistency of v_i is undisturbed because the error sources in $\mathcal{S}_i$, if there are any, are all incoming error sources which pass through v_i. Hence, v_i requires no further maintenance effort.

The intramodule error flow model just described models the error flow from the point of initial modification to other program areas utilizing the tracing functions. Due to the inherent complexity of these tracing functions, tracing intramodule error flow is a complicated process. This complexity inherent in the analysis of error flow can be circumvented, however, by an algorithmic solution to the error flow calculation. Instead of applying the tracing function on an individual path basis in order to build the propagation error source set of a block as the union of contributions of each path, the tracing function can be applied on a block-immediate successor block basis to form an algorithmic technique to calculate the error flow. Applying the tracing function on a block-immediate successor block basis means that errors are propagated from the initial error source block s to all immediate successor blocks v_i of s, and then from v_i to all immediate successor blocks of v_i, etc. Application of the tracing function in this manner builds the propagation error source set $\mathcal{S}_i$ of a block v_i in a stepwise manner with all the error sources which flow from an immediate predecessor block to v_i contributing to

the final $\mathcal{S}_i$. The tracing function is applied in this manner while new secondary error sources are created. After the error flow stabilizes, i.e. no new secondary error sources are created, the block identification criterion can then be applied as before on each block to determine if the block requires additional maintenance effort.

An algorithm for tracing this intramodule error flow and identifying the blocks which are affected by intramodule error flow has been developed. The input to this algorithm is a set $\mathcal{S}$ consisting of $(v', \mathcal{S}_{v'})$ such that v' is a primary error source block and $\mathcal{S}_{v'}$ is initialized as $LK_{v'}$. A set L is utilized to store the pairs $(v_i, \mathcal{S}_i)$ of blocks and their associated propagation error source sets. Another set L' is defined as a set consisting of the pair $(v_i, \mathcal{S}_i \cap C_i)$, where v_i is a block identified by the block identification criterion and $\mathcal{S}_i \cap C_i$ is the set of definitions in v_i which are affected by the error flow.

Algorithm 1. Intramodule Error Flow Tracing

Step 1. Initialize $L = \mathcal{S}$. For all $v_j \notin L$, set $\mathcal{S}_j = \emptyset$.

Step 2. If $L = \emptyset$, then go to Step 4. Otherwise, select an element $\ell' = (v', \mathcal{S}_{v'})$ in L, and set $L = L - \{\ell'\}$.

Step 3. For each $v_i \in I(v')$, if $f(v_i, \mathcal{S}_{v'}) \not\subseteq \mathcal{S}_i$, then set $\mathcal{S}_i = \mathcal{S}_i \cup f(v_i, \mathcal{S}_{v'})$ and $L = L \cup \{(v_i, \mathcal{S}_i)\}$. Go to Step 2.

Step 4. Initialize $L' = \emptyset$. For each $v_i \in V$, if $\mathcal{S}_i \cap C_i \neq \emptyset$, then set $L' = \{(v_i, \mathcal{S}_i \cap C_i)\} \cup L'$.

Theorem 1. Algorithm 1 identifies all of the pairs $(v_i, \mathcal{S}_i \cap C_i)$ of blocks and their error source sets which are affected by the intramodule error flow from the primary error sources in $\mathcal{S}$.

Proof. The proof of this theorem is straightforward. For the pair $(v_i, \mathcal{S}_i \cap C_i)$ to be in the intramodule error flow from the primary error sources in $\mathcal{S}$, then the following criteria must be satisfied:

1) $\mathcal{S}_i$ must be calculated for v_i by the intramodule error flow tracing functions previously defined, and
2) v_i may require further maintenance effort.

Now the tracing functions are applied on an individual path basis in order to build up the propagation error source set of a block as the union of the error flow contributions of each path. It is clear that Steps 1, 2, and 3 of

Algorithm 1 create an identical error source block applying the tracing functions on a block-immediate successor block basis instead of the union of paths approach, and hence criterion 1 is satisfied. Step 4 of Algorithm 1 then applies the block identification criterion to eliminate those blocks identified by Steps 1, 2, and 3 which do not require maintenance. Lemma 1 states the blocks satisfying the block identification criterion do not require maintenance, and hence criterion 2 is satisfied. Therefore, Steps 1, 2, 3, and 4 identify all blocks which are affected by the intramodule error flow.

3.1.2 Intermodule Error Flow Model

The intramodule error flow model just presented describes the error flow within program blocks and between program blocks in a module. In this section, an intermodule error flow model which models the error flow between modules will be presented. This intermodule error flow model characterizes the potential error flow properties of each module in the program so that the flow of error sources due to ripple effect can be emulated. To emulate the intermodule error flow, a basis must be defined to allow the tracing of the error flow within a module and across module boundaries. Two a priori conditions must exist before the intermodule error flow can occur. First, error sources must exist which have the capability to propagate between modules. Second, an enabled path must exist for error sources to be used during their propagation between modules. These paths consist of communication links (i.e. parameter passing and data sharing). The paths are enabled at the time of a module's invocation and, consequently, the invocation may have effects on both the invoked module and its surrounding environment.

In order to develop the intermodule error flow model, it is first necessary to develop a model for a large scale software system. A large scale software system can be considered as a collection of program modules. Let $\mathcal{M} = \{M_1, M_2, \ldots, M_j, \ldots M_n\}$ be the finite set of modules in the software system. There exists one and only one module in $\mathcal{M}$ which starts program execution upon invocation by the operating system. This module is called the initial entry module. Upon invocation, a module is executed and returns control to the invoking module. The set of modules which directly invoke a module M_j is denoted by $I^{-1}(M_j)$, and the set of modules which are directly invoked by M_j is denoted by $I(M_j)$.

A conceptually simple and direct way to model the intermodule error flow would be to view the system as one composite module. A composite module can be constructed by utilizing the invocation relationship among modules to perform inline code expansion; e.g. the code of a module M_j is inserted into M_k wherever M_k invokes M_j. The intermodule error flow within the software system could then easily be computed by the intramodule error flow algorithm. Although this technique is simple and direct, it has a severe limitation. If many modules are invoked many times, the physical size of the composite module would become very large and difficult to manage.

Another possibility for modelling the intermodule error flow utilizes a module error characteristics approach to characterize the potential error properties of each module in the software system. The module error characteristics of a module M_j are modelled by two sets $\mathcal{P}_j$ and $\mathcal{C}_j$ and the module level flow mapping $\bar{\mathcal{F}}_j(\mathcal{P}_j)$. Elements in $\mathcal{P}_j$ can propagate potential errors into M_j. Elements in $\mathcal{C}_j$ can cause potential errors to exist within M_j or flow out of M_j. The module level flow mapping $\bar{\mathcal{F}}_j(\mathcal{P}_j)$ maps each element p in $\mathcal{P}_j$ to a set of elements in $\mathcal{C}_j$ to denote that p can propagate potential errors to these elements in $\mathcal{C}_j$. The module error characteristics of a module have the same physical attributes as the block error characteristics of a block since the module error characteristics are defined from the block error characteristics of the local blocks in the module.

Since the intermodule error flow is enabled by module invocations, the potential error flow between the invoked and invoking modules must be established for each module invocation. Suppose that a module M_j is invoked in another module M_k. The potential error sources can flow to and from M_j only through the interface between M_j and other modules, i.e. the formal parameters of M_j and the data items shared between M_j and other modules. This potential error flow to and from M_j through M_j's interface can be modeled by M_j's error characteristics. To establish the potential error flow between M_j and M_k due to this invocation, a sequence of three blocks may be characterized for M_k. The first block in the sequence v_{i-1} is called an <u>input parameter error flow mapping block</u>, and is used to establish the potential error flow through the interface from M_k to M_j via input parameter passing. The second block in the sequence, v_i, is referred to as a <u>module invocation block</u>, and

is used to reflect the potential error flow to and from M_j through its interface. The third block in the sequence v_{i+1} is called an output parameter error flow mapping block, and is used to establish the potential error flow from M_j's interface back to M_k via output parameter passing. The block error characteristics of the input parameter error flow mapping block v_{i-1} are determined by input parameter mapping. Each data item x which appears as an actual input parameter can propagate potential error to its corresponding formal input parameter, say y. Hence, x is an element in P_{i-1}, y is an element in C_{i-1}, and y is mapped by x under $\bar{f}_{i-1}(P_{i-1})$. The block error characteristics of the module invocation block v_i are assigned the respective attributes of the module error characteristics of M_j, i.e. $P_i \leftarrow \mathcal{P}_j$, $C_i \leftarrow \mathcal{C}_j$, and $\bar{f}_i(P_i) \leftarrow \bar{\mathcal{F}}_j(\mathcal{P}_j)$. The block error characteristics of the output parameter error flow mapping block v_{i+1} are determined by output parameter mapping. Each formal output parameter w of M_j can propagate potential errors to its corresponding actual output parameter, say u. Hence, w is an element of P_{i+1}, u is an element in C_{i+1}, and u is mapped by w under $\bar{f}_{i+1}(P_{i+1})$. Note that if M_j has no formal parameters, only one block has to be assigned as the module invocation block for an invocation of M_j. If M_j is a function, the third block in the sequence can be omitted because the function name represents the sole output parameter of M_j and the function name is an element in M_j's interface which is referenced without parameter mapping.

Note that for a programming language, such as JOVIAL, which has the capability to syntactically identify a formal parameter as an input or output parameter, the error flow between the formal and actual parameters can be constructed before the module error characteristics of the invoked module are defined. Otherwise, it can only be constructed after the invoked module's error characteristics have been derived and the parameters have been classified as input and/or output parameters. A formal parameter of a module M_j is identified as a formal input parameter if it is an element in $\mathcal{P}_j$, or a formal output parameter if it is an element in $\mathcal{C}_j$.

The module error characteristics approach described above provides a natural way to characterize the error flow in a modular software system. Note that the block error characteristics of the block(s) assigned for a module invocation in the invoking module can only be specified after the

module error characteristics of the invoked module have been defined. Figure 2 illustrates the module error characteristics approach upon an invocation of M_j in M_k.

Modelling the intermodule error flow utilizing this module error characteristics approach requires that the potential error properties of the modules in $\mathcal{M}$ be characterized by their respective module error characteristics. It is necessary that a module have its error characteristics defined after the

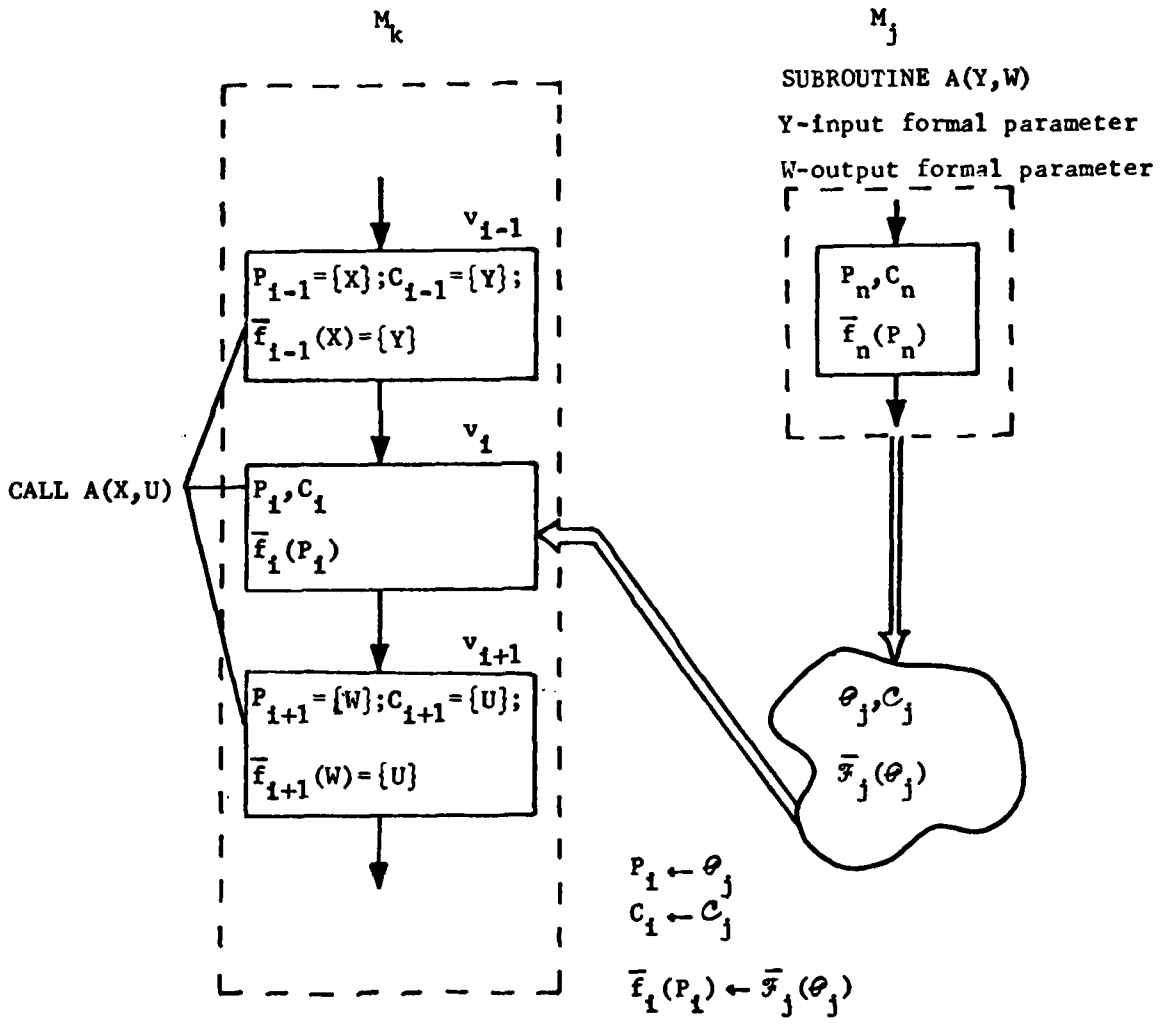


Figure 2. An illustration for the module error characteristics approach to intermodule error flow analysis.

module(s) it invokes because the block error characteristics of the block(s) assigned for a module invocation in the module can only be specified after the module error characteristics of the invoked module have been defined. The order in which the module error characteristics of the modules in $\mathcal{M}$ are defined is thus determined by the reverse invocation order [12]. However, there is a limitation imposed by the reverse invocation order. That is, no module can be invoked recursively. Although this limitation can be overcome, we will assume that no modules are invoked recursively.

Since the intermodule error flow is modelled utilizing a module error characteristics approach, it is necessary to analyze the computation of these module error characteristics. This requires an examination of the properties of the intermodule error flow. Since the intermodule error flow is enabled at the time of a module's invocation, potential error sources can flow to and from a module via parameter passing and data sharing. The formal parameters of a module and the data items which are shared between the module and other modules form the parameter list of the module, which defines the interface between the module and its surrounding environment. An error source can propagate to and from a module *if and only if the affected data item* is an element of the parameter list of the module. Thus, an element in the parameter list of a module possesses a passed attribute or a global attribute depending upon the scope of effect of the data item. Figure 3 illustrates the concept of parameter list through which potential errors can flow across the module's boundaries.

Elements in a module's parameter list possess certain error properties which can be passive and active. A passive element is a data item that is in the module's parameter list, but cannot cause inconsistency to exist within the module. An example of this type of element would be a data item which is just passing through the module. Thus, if this data item is an error source, it cannot create new error sources within the module since the module does not make any direct use of the data item. An active element is a data item which can cause inconsistency to exist within the module. This element possesses the following error properties.

1) It can be an incoming error source which causes the creation of a potential error source which can flow out of the module.

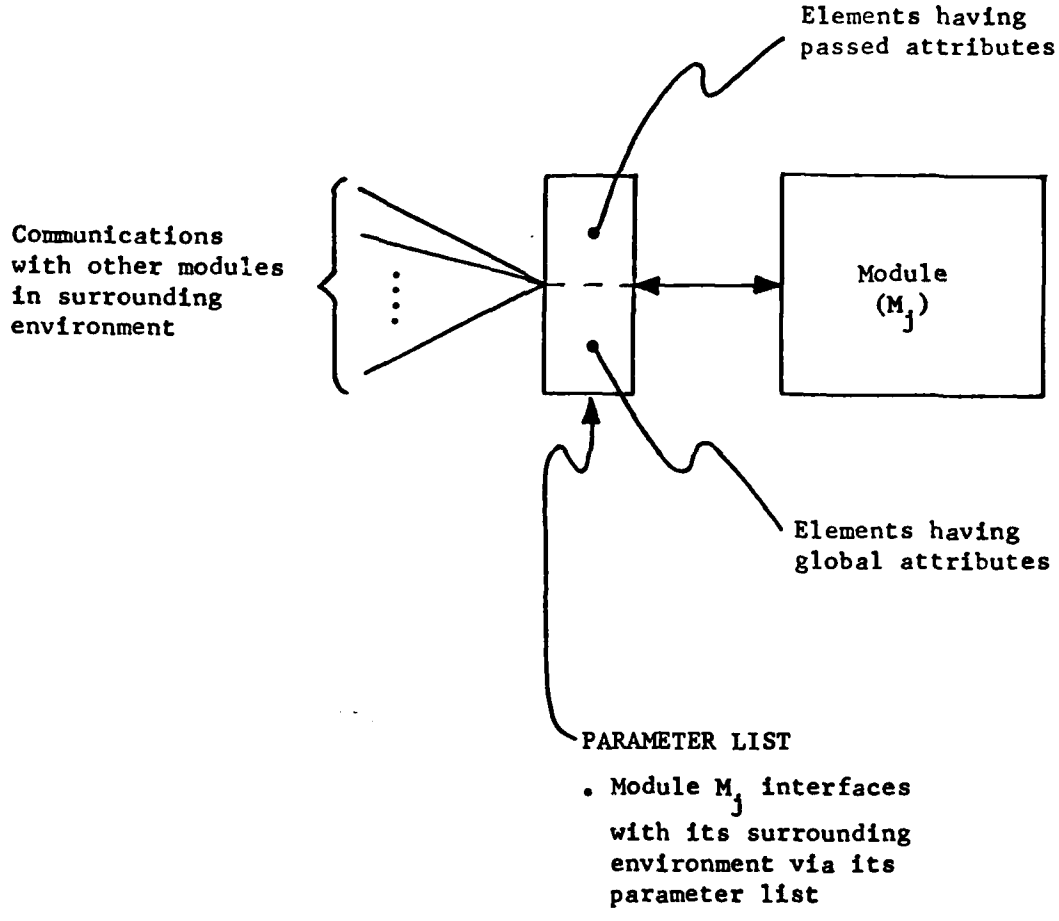
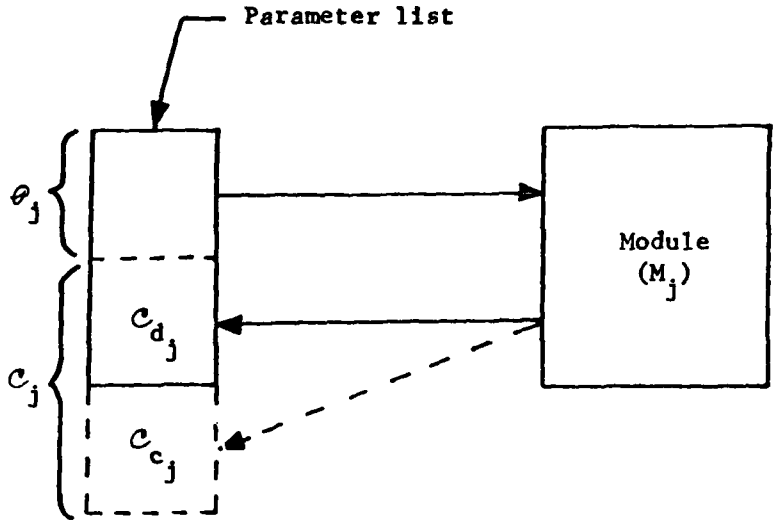


Figure 3. An illustration for operational environment model.

2) It can be an incoming error source which causes the creation of potential error sources which all remain within the module.
3) It can be an error source which flows out of the module.
4) It can be a combination of 1) and 3), or 2) and 3).

Utilizing the error properties that each element in the parameter list of a module may possess, the potential error properties of a module which are modelled by the module error characteristics of the module can be characterized. Let M_j denote the module being characterized. The elements in Θ_j which can propagate potential errors to M_j possess error property 1) or 2). $\mathcal{C}_j$ is partitioned into the subsets $\mathcal{C}_{d_j}$ and $\mathcal{C}_{c_j}$. Subset $\mathcal{C}_{d_j}$ represents the set of natural source capable definitions which can propagate potential errors out of M_j. The elements in $\mathcal{C}_{d_j}$ possess the error property 3). An element p in Θ_j which possesses the error property 1) maps to a set of elements in $\mathcal{C}_{d_j}$ under $\overline{\mathcal{F}}_j(\Theta_j)$. This implies that p can propagate potential errors to these elements in $\mathcal{C}_{d_j}$ which can flow out of M_j. The subset $\mathcal{C}_{c_j}$ consists of pseudo source capable definitions which are defined for the elements in Θ_j, each of which possesses the error property 2). The elements in $\mathcal{C}_{c_j}$ cannot propagate potential errors out of M_j. The existence of $\mathcal{C}_{c_j}$ is required to preserve the compatibility between the error characteristics of a module and the error characteristics of a block. Preservation of the compatibility insures that the intramodule error flow algorithm will correctly identify those local blocks which are affected by the error flow and invoke M_j. The elements in $\mathcal{C}_{c_j}$ can be defined in such a manner that they will not create any erroneous secondary error sources within the invoking module. The element in $\mathcal{C}_{c_j}$ assigned for an element in Θ_j which possesses the error property 2) is mapped by that element in Θ_j under $\overline{\mathcal{F}}_j(\Theta_j)$. Figure 4 illustrates the concept of module error characteristics which characterize the potential error properties of the modules.

Essential to computing a module's error characteristics is the assignment of a dummy entry block and a dummy exit block for each module. The entry block which represents the single entry point of the module is used to accommodate the flow of error sources from other modules to the module. The exit block which represents the single exit point of the module is used to accommodate the flow of error sources out of the module. The block error characteristic sets of the two blocks are all specified to be empty, and hence the



$\mathcal{P}_j$ - module level potential propagator set

$\mathcal{C}_j$ - module level source capable set

$\mathcal{C}_{d_j}$ - natural source capable definition set

$\mathcal{C}_{c_j}$ - pseudo source capable definition set

$\overline{\mathcal{F}}_j(\mathcal{P}_j)$ - module level flow mapping

Figure 4. An illustration for error environment model.

flow mappings are undefined.

A procedure can be defined to compute the module error characteristics of a module M_j. Since the parameter list of M_j expresses the interface between M_j and its surrounding environment, the procedure starts with identifying the parameter list of M_j.

Definition 7. The passed parameter set P_j^* of a module M_j consists of all the data items which appear in the formal parameter list of M_j and the function name if M_j is a function.

Definition 8. The global parameter set $\mathcal{D}_j$ of a module M_j consists of all the data items referenced in M_j, each of which has a scope of effect over M_j.

For a module which invokes other modules, the global parameter set of the module must be augmented to include the passive elements in the parameter

list.

Definition 9. The augmented global parameter set $\mathcal{B}_j$ of a module M_j is defined as $\mathcal{B}_j \cup [\bigcup_{M_s \in I(M_j)} \mathcal{B}_s]$, where each $x \in \{[\bigcup_{M_s \in I(M_j)} \mathcal{B}_s] - \mathcal{B}_j\}$ is a passive element which possesses a global scope of effect over M_j and/or its successors.

Definition 10. The parameter list I_j of a module M_j is defined as $P_j^* \cup \mathcal{B}_j$ which is the interface between M_j and its environment.

An element α in I_j can propagate potential errors out of M_j and hence is an element in $\mathcal{C}_{d_j}$ if α has been defined within M_j. The following definition identifies the set $\mathcal{C}_{d_j}$, the set of elements in I_j which possess the error property 3).

Definition 11. The natural source capable definition set $\mathcal{C}_{d_j}$ of a module M_j is defined as $I_j \cap [\bigcup_{v_i \in V_j} C_i]$, where V_j denotes the vertex set of the program graph of M_j.

An element α in I_j may propagate potential errors into M_j, and hence is a potential propagator candidate if the element has been used within M_j.

Definition 12. The potential propagator candidate set $\mathcal{U}_j$ is defined as $I_j \cap [\bigcup_{v_i \in V_j} P_i]$, where V_j denotes the vertex set of the program graph of M_j.

To determine if an element α in $\mathcal{U}_j$ possesses the error property 1) or 2), α is treated as a primary error source and is the only element of $\mathcal{S}_e$, where $\mathcal{S}_e$ is the propagation error source set of the entry block v_e of M_j. Then, we compute the intramodule error flow in M_j, and examine the propagation error source set $\mathcal{S}_f$, where $\mathcal{S}_f$ is the propagation error source set of the exit block v_f of M_j. If the set $\mathcal{S}_f$ is empty, α cannot propagate potential errors into M_j. However, if the set $\mathcal{S}_f$ is not empty, then α can propagate potential errors into M_j. Furthermore, if $\mathcal{C}_{d_j} \cap \mathcal{S}_f$ is not empty, α can propagate potential errors to the elements in $\mathcal{C}_{d_j} \cap \mathcal{S}_f$, which in turn flow out of M_j and hence α possesses the error property 1). Otherwise, if $\mathcal{C}_{d_j} \cap \mathcal{S}_f$ is empty, α can propagate potential errors to the elements in $\mathcal{S}_f$. But, all the elements remain within M_j, and hence α possesses the error property 2). The following definitions describe this process:

Definition 13. Let $S = \mathcal{U}_j \cup [\bigcup_{v_i \in V_j} C_i]$, which denotes the set of all definitions and potential propagator candidates of M_j. Let S^* denote the set of all subsets of S. For each element α in $\mathcal{U}_j$, the potential propagator

identification function $h: \mathcal{U}_j \rightarrow S^*$ is defined by

$$h(\alpha) = \{\bigcup_p F_f(p) \mid p \text{ is a valid path from } v_e \text{ to } v_f\}, \tag{9}$$

where

$$F_f(p) = \{f(v_f, f(v_{f-1}, \ldots, f(v_1, \mathcal{S}_e = \{\alpha\}), \ldots)) \mid p = v_e, v_1, \ldots, v_f\} \tag{10}$$

Definition 14. The natural potential propagator set $\mathcal{P}_{d_j}$ of a module M_j consists of the elements in the potential propagator candidate set $\mathcal{U}_j$, in which each element α satisfies the condition that $\mathcal{C}_{d_j} \cap h(\alpha) \neq \emptyset$.

The elements in $\mathcal{P}_{d_j}$ thus possess the error property 1).

Definition 15. The pseudo potential propagator set $\mathcal{P}_{c_j}$ of a module M_j consists of the elements in the potential propagator candidate set $\mathcal{U}_j$, in which each element α satisfies the conditions that $h(\alpha) \neq \emptyset$ and $\mathcal{C}_{d_j} \cap h(\alpha) = \emptyset$.

The elements in $\mathcal{P}_{c_j}$ thus possess the error property 2).

Definition 16. The module level potential propagator set $\mathcal{P}_j$ of a module M_j is defined as $\mathcal{P}_{c_j} \cup \mathcal{P}_{d_j}$.

The elements in $\mathcal{P}_j$ possess the error property 1) or 2) and hence they can propagate potential errors into M_j.

Recall that a pseudo source capable definition must be assigned for each element in $\mathcal{P}_{c_j}$ in order to preserve the compatibility between the error characteristics of a module and the error characteristics of a block. The assignment of a pseudo source capable definition must insure that it will not create any erroneous secondary error sources within the invoking module.

Definition 17. The pseudo source capable definition set $\mathcal{C}_{c_j}$ is defined as $\{g(\alpha) \mid \alpha \in \mathcal{P}_{c_j}\}$, where g is any arbitrary function which maps α into an element in $\mathcal{C}_{c_j}$ in such a manner that if M_k invokes the set of modules $I(M_k)$, then

$$[\bigcup_{M_j \in I(M_k)} \mathcal{C}_{c_j}] \cap [\bigcup_{v_i \in V_k} (C_i \cup P_i)] = \emptyset \tag{11}$$

and

$$\bigcap_{M_j \in I(M_k)} \mathcal{C}_{c_j} = \emptyset. \tag{12}$$

The function g can be realized by assigning a control item to represent the pseudo source capable definition required for a pseudo potential

propagator. Thus, the two conditions (11) and (12) can be automatically satisfied and hence no checking is required.

Definition 18. The module level source capable set $\mathcal{C}_j$ is defined as $\mathcal{C}_{d_j} \cup \mathcal{C}_{c_j}$.

The elements in $\mathcal{C}_j$ can cause potential errors to exist within M_j or flow out of M_j. Note that the elements in $\mathcal{C}_j$ can also be partitioned into two subsets $\mathcal{C}_{r_j}$ and $\bar{\mathcal{C}}_{r_j}$ according to their respective addressing capabilities. Hence, when $\mathcal{C}_j$ is assigned to C_i of a block v_i in a module which invokes M_j, the two subsets of $\mathcal{C}_j$ are assigned to their corresponding subsets of C_i, i.e. $C_{r_i} \leftarrow \mathcal{C}_{r_j}$, and $\bar{C}_{r_i} \leftarrow \bar{\mathcal{C}}_{r_j}$.

Definition 19. The module level flow mapping $\bar{\mathcal{F}}_j: \mathcal{\Theta}_j \rightarrow \mathcal{C}_j$ of a module M_j maps each element p in $\mathcal{\Theta}_j$ of M_j to a set of elements in $\mathcal{C}_j$ of M_j to denote that p can propagate potential errors to these elements in $\mathcal{C}_j$. The subset in $\mathcal{C}_j$, in which each element is an image of the element p in $\mathcal{\Theta}_j$ through the mapping $\bar{\mathcal{F}}_j$, is denoted by $\bar{\mathcal{F}}_j(p)$. Similarly, the subset in $\mathcal{C}_j$, in which each element is an image of some elements in $\mathcal{\Theta}_j$ through the mapping $\bar{\mathcal{F}}_j$, is denoted by $\bar{\mathcal{F}}_j(\mathcal{\Theta}_j)$, i.e., $\bar{\mathcal{F}}_j(\mathcal{\Theta}_j) = \bigcup_{p \in \mathcal{\Theta}_j} \bar{\mathcal{F}}_j(p)$.

Based on Definitions 14, 15, and 17, the module level flow mapping on each element p in $\mathcal{\Theta}_j$ can be identified by the following two rules:

$$\bar{\mathcal{F}}_j(p) = \mathcal{C}_{d_j} \cap h(p) \quad \text{for } p \in \mathcal{\Theta}_{d_j}, \tag{13}$$

$$\bar{\mathcal{F}}_j(p) = \{g(p)\} \quad \text{for } p \in \mathcal{\Theta}_{c_j}, \tag{14}$$

where $\mathcal{\Theta}_j = \mathcal{\Theta}_{c_j} \cup \mathcal{\Theta}_{d_j}$ and $\mathcal{\Theta}_{c_j} \cap \mathcal{\Theta}_{d_j} = \emptyset$.

Derivation of the module error characteristics of a module M_j requires identification of the parameter list I_j of M_j. Given the parameter list I_j, the block error characteristics of all blocks in M_j, and the program graph of M_j, the module error characteristics of M_j which are represented by $\mathcal{\Theta}_j$, $\mathcal{C}_j$, and $\bar{\mathcal{F}}_j(\mathcal{\Theta}_j)$ can be derived by the following algorithm. The algorithm is based on Definitions 11-19 which describe the properties and derivations of the various error sets and mappings.

Algorithm 2. Algorithmic Module Error Characteristics Identification

Step 1. Initialize the sets $\mathcal{\Theta}_j$ and $\mathcal{C}_j$ to be empty, and $\bar{\mathcal{F}}_j(\mathcal{\Theta}_j)$ is undefined.

Step 2. Create the set $\mathcal{C}_{d_j}$ by computing $I_j \cap [\bigcup_{v_i \in V_j} C_i]$. Assign $\mathcal{C}_{d_j}$ to $\mathcal{C}_j$.

Step 3. Create the set $\mathcal{U}_j$ by computing $I_j \cap [\bigcup_{v_i \in V_j} P_i]$.

Step 4. For each element α in $\mathcal{U}_j$, let v_e be the only primary error source block and α be the only element in $\mathcal{E}_e$. Trace intramodule error flow in M_j as described in Algorithm 1. When the intramodule error flow in M_j stabilizes, check $\mathcal{E}_f$. If $\mathcal{E}_f \neq \emptyset$ and $\mathcal{E}_f \cap \mathcal{C}_{d_j} \neq \emptyset$, add α to Θ_j and set $\overline{\mathcal{F}}_j(\alpha) = \mathcal{E}_f \cap \mathcal{C}_{d_j}$. If $\mathcal{E}_f \cap \mathcal{C}_{d_j} = \emptyset$ but $\mathcal{E}_f \cap \mathcal{C}_{d_j} = \emptyset$, add α into Θ_j and generate a control item. Add this control item to $\mathcal{C}_j$ and set $\overline{\mathcal{F}}_j(\alpha)$ to contain the control item as the only element. Continue processing the elements in $\mathcal{U}_j$. When all are processed, terminate.

It is clear that Algorithm 2 correctly identifies the module error characteristics of a module based on Definitions 11-19.

The identification of module error characteristics accomplished by Algorithm 2 in conjunction with the intramodule error flow provide the basis for the calculation of the intermodule error flow. The intramodule error flow model tracks the flow of potential errors within a module. The concept of block error characteristics as given by Definitions 1-3 in conjunction with the concept of module error characteristics as given by Definitions 16, 18, and 19 underlie the ability of the intramodule error flow model to track the potential error flow within a program. Thus, tracing the intramodule error flow depends on the ability to track the flow of error sources across module boundaries. This flow of error sources between modules constitutes the intermodule error flow. Potential errors can propagate from a module M_j to the modules which invoke M_j and the modules which are invoked by M_j. When there exists the error flow from M_j to the modules invoked by M_j, potential errors are said to propagate in a <u>downward</u> direction with respect to M_j. Similarly, when there exists the error flow from M_j to the modules which invoke M_j, potential errors are said to propagate in an <u>upward</u> direction with respect to M_j. It is apparent that the downward intermodule error flow with respect to M_j must be identified before the upward intermodule error flow with respect to M_j is identified; otherwise, the latter cannot be completely characterized.

The following lemma states the necessary criterion to determine the presence of downward intermodule error flow from M_j to M_k upon an invocation of M_k in M_j.

<u>Lemma 2</u>. (<u>Criterion for Downward Intermodule Error Flow</u>). Suppose that M_k is

invoked in M_j and v_ℓ is the module invocation block assigned in M_j for this module invocation. Given the propagation error source set $\mathcal{E}_\ell$, M_j can propagate potential errors to M_k via this module invocation if $\mathcal{E}_\ell \cap \mathcal{C}_k \neq \emptyset$, where $\mathcal{C}_k$ is the module level source capable set of M_k.

Proof. $\mathcal{E}_\ell \cap \mathcal{C}_k = \emptyset$ implies that v_ℓ is incapable of internally generating secondary error sources as a direct result of the intramodule error flow. Therefore, all error sources contained in $\mathcal{E}_\ell$ must pass through v_ℓ without propagating to M_k. Thus, $\mathcal{E}_\ell \cap \mathcal{C}_k \neq \emptyset$ is a sufficient criterion to determine the presence of the downward intermodule error flow from M_j to M_k upon this module invocation.

Error sources which propagate from M_j to M_k due to the downward intermodule error flow are specified as the downward primary error sources of M_k. A downward primary error source can be formally defined as follows:

Definition 20. Let M_k be invoked by M_j, and v_ℓ be the module invocation block in M_j for this module invocation. Let $\mathcal{E}_e$ denote the propagation error source set of the entry block v_e in M_k. Given the propagation error source set $\mathcal{E}_i$ of each block v_i in M_j, where $v_i \in I^{-1}(v_\ell)$, a data item x is a downward primary error source and is added to LK_e if $x \in [\bigcup_{v_i \in I^{-1}(v_\ell)} \mathcal{E}_i] \cap \mathcal{P}_k$, where $\mathcal{P}_k$ is the module level potential propagator set of M_k. Downward primary error sources are utilized to identify the secondary error sources within a module which is invoked by a module that is affected by the error flow.

In order to compute the complete intermodule error flow, the propagation of error sources caused by the downward intermodule error flow must be supplemented with the propagation of error sources caused by the upward intermodule error flow. The following lemma defines the necessary criterion which determines the presence of the upward intermodule error flow from M_k to an invoking module upon an invocation of M_k.

Lemma 3. (Criterion for Upward Intermodule Error Flow). Given the propagation error source set $\mathcal{E}_f$ of the exit block v_f in a module M_k, M_k can propagate potential errors from itself to any module M_j which directly invokes M_k if $\mathcal{E}_f \cap \mathcal{C}_{d_k} \neq \emptyset$, where $\mathcal{C}_{d_k}$ is the natural source capable definition set of M_k.

Proof. $\mathcal{E}_f \cap \mathcal{C}_{d_k} = \emptyset$ implies that M_k is incapable of propagating error sources. Therefore, all error sources contained in $\mathcal{E}_f$, if there are any,

remain within M_k without propagating to M_j. Thus, $\mathcal{S}_f \cap \mathcal{C}_{d_k} \neq \emptyset$ is a sufficient criterion to determine the presence of the upward intermodule error flow from M_k to M_j.

Error sources which propagate from M_k due to the upward intermodule error flow are upward primary error sources of M_j, where $M_j \in I^{-1}(M_k)$. An upward primary error source can be formally defined as follows:

Definition 21. Let M_k be invoked by M_j, and v_i be the module invocation block assigned in M_j for this module invocation. Let $\mathcal{S}_i$ denote the propagation error source set of v_i. Given the propagation error source set $\mathcal{S}_f$ of the exit block v_f in M_k, a data item x is an upward primary source and is added to LK_i if $x \in \mathcal{S}_f \cap \mathcal{C}_{d_k}$.

Given the set $\bar{m}$ of modules which are involved in the initial modification and the set $\mathcal{S}_j$ which consists of the blocks and their associated primary error sources in each module in $\bar{m}$, an algorithm can be developed for tracing the error flow from the initial modification to the other portions of the program affected by the modification. Thus, this algorithm will trace the error flow as a consequence of the modification. To accomplish this objective, the algorithm must utilize the models of the intramodule error flow, the downward intermodule error flow and the upward intermodule error flow previously described. The intramodule error flow is utilized to trace the error flow within a module to its exit point. The downward intermodule error flow is then utilized to trace the error flow in invoked modules. The upward intermodule error flow is then utilized to trace the error flow in invoking modules. Tracing continues until the error flow stabilizes, i.e. no new error sources are created.

Algorithm 3. Error Flow Tracing

Step 1. Define a set $m^{\&} = \bar{m}$. $m^{\&}$ will contain the set of modules potentially affected by the upward intermodule error flow. Define another set, $m^* = \emptyset$, which will be utilized to contain the modules affected by the error flow. For each module M_j in the program, the propagation error source set of the entry block in M_j and the set L_j are initialized to be empty, where L_j consists of the blocks in M_j and their associated propagation error source sets.

Step 2. If $\bar{m}$ is empty, go to Step 4; otherwise, select a module from $\bar{m}$ and delete it from $\bar{m}$. Let M_j denote the selected module. Apply the intramodule

error flow tracing algorithm to trace the intramodule error flow in M_j. Add the blocks and their associated error sources identified by the block identification criterion into L_j'.

Step 3. For each block v_i contained in L_j', check if it is a module invocation block. If v_i is a module invocation block assigned for an invocation to M_k, calculate the set of error sources $\bigcup_{v' \in I^{-1}(v_i)} \mathcal{E}_{v'}$, which currently flow into M_k. Then, check if $(\mathcal{E}_e)_{M_k}$ is properly contained in $(\bigcup_{v' \in I^{-1}(v_i)} \mathcal{E}_{v'})_{M_j} \cap \mathcal{C}_k$. If it is, i.e., new error sources flow into M_k, then add M_k into $\mathcal{M}^*$ and $\bar{\mathcal{M}}$. Furthermore, the entry block is added into $\mathcal{L}_k$ while the primary error source set of the entry block is updated by $(\mathcal{E}_e)_{M_k} \cup [(\bigcup_{v' \in I^{-1}(v_i)} \mathcal{E}_{v'})_{M_j} \cap \mathcal{C}_k]$. Continue processing the elements in $\bar{\mathcal{M}}$. When all are processed, go to Step 4.

Step 4. If $\mathcal{M}^{\&}$ is empty, i.e., the error flow stabilizes, then terminate. At termination, $\mathcal{M}^*$ contains all of the modules affected by the error flow, and the L_j' sets of the modules in $\mathcal{M}^*$ contain the blocks and their associated error sources identified by the block identification criterion.

If $\mathcal{M}^{\&}$ is not empty, then for each module in $\mathcal{M}^{\&}$, apply the upward intermodule error flow criterion by letting M_j be a member of $\mathcal{M}^{\&}$ and calculating $(\mathcal{E}_f)_{M_j} \cap \mathcal{C}_{d_j}$. If the intersection is empty, i.e. no error sources currently flow from M_j, then examine another module in $\mathcal{M}^{\&}$. Otherwise, add the modules which directly invoke M_j into $\bar{\mathcal{M}}$ and $\mathcal{M}^*$. Furthermore, for each module M_k which invokes M_j, add the blocks in M_k which are module invocation blocks assigned for invocations to M_j into $\mathcal{L}_k$ and update the sets of primary error sources of these blocks by $(\mathcal{E}_f)_{M_j} \cap \mathcal{C}_{d_j}$.

After all the modules in $\mathcal{M}^{\&}$ have been examined, assign $\bar{\mathcal{M}}$ to $\mathcal{M}^{\&}$ and branch to Step 2 to identify the net effect on the modules currently in $\mathcal{M}^{\&}$ and the modules invoked by the members in $\mathcal{M}^{\&}$.

Theorem 2. Algorithm 3 traces the error flow as a consequence of a modification.

Proof. The proof of this theorem is straightforward. Tracing error flow consists of both upward and downward error flow tracing. The algorithm utilizes the upward and downward error flow models previously developed for tracing error flow among modules. Thus, the algorithm simply consists of repeated

applications of the intramodule and intermodule error flow models until the error flow stabilizes. It is noted that the error flow must stabilize since the number of program modules and blocks is finite, the number of data and control items is finite, and the propagation error source set of each block is of finite size. Hence, to show that Algorithm 3 terminates, it is sufficient to show that the algorithm terminates when the error flow stabilizes.

When the error flow stabilizes, the execution of Steps 2 and 3 will make $\overline{m}$ empty since no new downward primary error sources can be created by the modules originally in $\overline{m}$. Then, the execution of Step 4 will keep $\overline{m}$ empty since no new upward primary error sources can be created by the modules in $m^{\&}$. At the end of Step 4, $m^{\&}$ will also become empty since $m^{\&}$ will be updated by $\overline{m}$ which is empty. The execution will then branch to Step 2 which will first check if $\overline{m}$ is empty. Since $\overline{m}$ is empty now, the next step executed will be Step 4. Because now we have the condition that $m^{\&}$ is empty, the algorithm will terminate. This completes the proof of the theorem.

3.2 Logical Ripple Effect Analysis Technique

In the previous sections, logical ripple effect has been analyzed through the development of an intramodule error flow model and an intermodule error flow model. In this section, a logical ripple effect analysis technique based upon these models will be described. The logical ripple effect analysis technique consists of two phases.

The first phase is the lexical analysis phase which is performed on the program upon completion of the initial modification. The lexical analysis characterizes the potential error properties of the modified program. In order to reduce the amount of effort required to develop the lexical analyzers for several high-level programming languages, lexical analysis can be performed in two steps. The first step is language dependent. During this step different syntactic and semantic constructs unique to the programming language utilized are analyzed in order to define program blocks, the control flow between program blocks, the error characteristics of the program blocks, etc. in a universal format which can be processed by step two of the lexical analysis. The main function of step two of the lexical analysis phase is to utilize the output of step one in order to compute the error characteristics of the modules in the program, and hence characterize the potential error

properties of the program.

The second phase of the ripple effect analysis technique consists of tracing the logical ripple effect as a consequence of the modification. The input to the second phase consists of the output of the lexical analysis phase in addition to the primary error sources involved in the initial modification.

3.2.1 Lexical Analysis Step One

In this section, step one of lexical analysis for the logical ripple effect analysis technique will be described. Two assumptions imposed by the error flow analysis models concerning data items must first be discussed. The assumptions are that each data item possesses a unique memory address and that each memory address corresponds to a unique data item. These assumptions are made because the error flow analysis models treat memory in only a symbolic sense, while most programming languages permit the programmer to declare data items with the same data name, but different scopes of effect. This capability can introduce address aliasing. One way to solve this problem is to keep track of the scopes of effect for all data items. When the data name is referenced in the program, the data item with the scope of effect for the reference can be resolved. The reference can then be relabelled to reflect which data item has the applicable scope of effect for the reference. Also, some programming languages allow the user to declare several data names for the same data item. The EQUIVALENCE statement in FORTRAN is an example. This capability can introduce symbolic aliasing. This kind of symbolic aliases can be identified by seeking out the syntactic constructs used by each language to define the alias relation. Once identified, the symbolic aliasing can be resolved by substituting only one element in the alias grouping for all other elements in the group throughout the scope of effect of the aliases. Thus, it is always possible to resolve all address aliasing and symbolic aliasing, and hence satisfy the two assumptions for the error flow analysis models.

In this step, schemes, that are unique to each programming language to derive the program graphs for the modules in the program and to identify the definitions and usages from all types of statements and expressions, must be established according to the syntactic and semantic constructs of the programming language. During this step, a control item is assigned to represent each control directive which is defined by a conditional or iterative statement.

The items appearing in the control directive are treated as usages in order to define the control item. In addition, an iterative statement, such as a DO statement in FORTRAN, is assigned a block by itself. When the loop variable is referenced within the scope of the loop, the control item assigned to represent the control directive of the loop is treated as a usage instead of the loop variable. Furthermore. the items which are utilized in evaluating the index or pointer value when an element in a data structure is referenced employing implicit addressing are always treated as usages because they can affect which element in the data structure is referenced.

During this step, module invocations in a module must also be identified. A sequence of blocks is assigned in the invoking module for each module invocation. The number of blocks in the sequence is determined by the type and the formal parameter declaration of the invoked module. The error characteristics of the block(s) in the sequence are not specified in this step. But, the actual parameter list which appears in this module invocation is stored in conjunction with the module invocation. For each module M_j in the program, the set $\mathcal{I}_j$ is defined from this module invocation information. Each element in $\mathcal{I}_j$ is an ordered triplet $(v_i, M_k, \bar{P}_{v_i})$, where v_i is the module invocation block in M_j for an invocation of M_k, and $\bar{P}_{v_i}$ is the set of blocks which are used to establish the error flow between input or output parameters. The connectivity relationship between modules in the program which completely characterizes the call graph [12] is also constructed from the module invocation information in each module. The passed parameter set and global parameter set of each module are also constructed as a by-product.

During this step, the block error characteristics of a block must also be identified. Upon entering a block v_i, the two sets P_i and C_i are initialized to be empty and hence $\bar{f}_i(P_i)$ is undefined. When the end of each statement in v_i is reached, an intrablock potential error flow identification algorithm is invoked to derive the intrablock potential error flow within v_i based on the definitions and usages identified from the statement. By statement, reference is meant to the semantic construct instead of the syntactic construct. For example, the FORTRAN logical IF statement, IF(A.LT.B)A = A + 1 is treated as two statements: IF(A.LT.B) and A = A + 1. Upon exiting from the block v_i, a block error characteristics identification algorithm is

invoked to derive the sets P_i and C_i, and the flow mapping $\bar{f}_i(P_i)$ based on the intrablock potential error flow within v_i. Note that the intrablock potential error flow identification algorithm must operate on a statement-by-statement basis in order to accommodate the appearance of a function reference within a statement. The process to establish the potential error flow between the invoking module and the invoked function must be carried out before the potential error property of the statement containing the function reference is identified. Hence, the scheme to identify the intrablock potential error flow of a block must operate on a statement-by-statement basis in order to handle this two-step process for a function reference. Thus, in this scheme, when a function reference is identified in a statement, the function name is treated as a usage in order to define the relevant definition(s) while the actual input parameter list is stored in conjunction with the function reference. A sequence of blocks as previously described for module invocations is then assigned for the function reference to establish the potential error flow between the invoking module and the invoked function. Another new block which is an immediate successor of the module invocation block specified for the function reference is then entered. Upon reaching the end of the statement which contains the function reference, the intrablock potential error flow algorithm is then invoked to process the definitions and usages identified from the statement.

In this step, it must also be determined whether or not an element in a data structure has been used or defined in a block. Thus, the index or pointer value in each reference to an element in a data structure must be maintained in a symbolic fashion. Note that a scheme must be devised to correctly maintain the symbolic value of the index or pointer value in each reference to an element in a data structure within a block. For example, consider the case that, in block v_i, A(J) is referenced first and then J is incremented by one. Later, when A(J-1) is referenced in v_i, we should be able to tell that the two references are made to the same element in the data structure A. If x is a data structure whose elements employ implicit addressing, the element in x which is referenced by an index or pointer value $\underline{x}$ is denoted as $x'\underline{x}$. For the sake of consistency, a data or control item y which employs explicit addressing is also denoted as $y'\underline{y}$, except that the index $\underline{y}$

is assigned a character constant which denotes that y employs explicit addressing.

A reference set R is defined to store the information about the data or control items which have been referenced in a block. When a block is entered, the set R is initialized to be empty. Each element in R is then represented as an ordered triplet ($r'\underline{r}$, $DB_{r'\underline{r}}$, $UB_{r'\underline{r}}$), where $r'\underline{r}$ is a control or data item which has been referenced in the block, $DB_{r'\underline{r}}$ is the set of data or control items which can propagate potential errors to $r'\underline{r}$, and $UB_{r'\underline{r}}$ is the set of data or control items to which $r'\underline{r}$ can propagate potential errors from outside of v_i. For each statement in the block, a definition-usage set DU is constructed from the definitions and usages identified from the statement. Each element in DU can be expressed as an ordered pair ($d'\underline{d}$, $U_{d'\underline{d}}$) where $d'\underline{d}$ is a definition and $U_{d'\underline{d}}$ is the set of usages which are utilized to define $d'\underline{d}$.

It is noted that constants, e.g. numerical, Boolean, or character constants, are not considered in our definition as data or control items. However, a data item may be defined by an expression in which only constants and/or operators are involved. In this case, the usage set associated with the data item would be empty. In order to denote that a data item has been defined by an expression which involves only constants and/or operators we use a pseudo item of the form $c'\underline{c}$, where $\underline{c}$ is a unified character constant denoting that the item is a constant. This pseudo item will be used in the description of the two algorithms to identify the error characteristics of the blocks.

Given the set DU constructed for a statement contained in a block v_i, and the partially constructed set R for v_i, the intrablock potential error flow can be identified by the following algorithm:

Algorithm 4. Intrablock Error Flow Identification Algorithm

Step 1. If the set DU is empty, then terminate; otherwise, select an element from DU. Let ($d'\underline{d}$, $U_{d'\underline{d}}$) be the element selected from DU. Delete the element from DU.

Step 2. Check if $d'\underline{d} \in U_{d'\underline{d}}$. If $d'\underline{d} \in U_{d'\underline{d}}$, set the flag f_1 to be 'true'; otherwise, set f_1 to be 'false'.

Step 3. Search for an element ($d'\underline{d}$, 'x', 'x') in R, where 'x' denotes 'don't care'. If ($d'\underline{d}$, 'x', 'x') $\in$ R, go to Step 5.

Step 4. If f_1 = 'true', then add (d'$\underline{d}$, {d'$\underline{d}$}, {d'$\underline{d}$}) to R; otherwise, add (d'$\underline{d}$, $\emptyset$, $\emptyset$) to R. Go to Step 6.

Step 5. If $DB_{d'\underline{d}} = \emptyset$ and f_1 = 'true', then add d'$\underline{d}$ to both sets $DB_{d'\underline{d}}$ and $UB_{d'\underline{d}}$. If $DB_{d'\underline{d}} \neq \emptyset$ and f_1 = 'false', then delete y'$\underline{y}$ from $DB_{d'\underline{d}}$ and also delete d'$\underline{d}$ from the set $UB_{y'\underline{y}}$, for each element y'$\underline{y}$ in $DB_{d'\underline{d}}$ such that $\underline{y} \neq \underline{c}$.

Step 6. If $U_{d'\underline{d}} = \emptyset$, then add c'$\underline{c}$ to $DB_{d'\underline{d}}$ and go to Step 1. If f_1 = 'true', then delete d'$\underline{d}$ from $U_{d'\underline{d}}$.

Step 7. If $U_{d'\underline{d}} = \emptyset$, then branch to Step 1; otherwise, select an element from $U_{d'\underline{d}}$. Let u'$\underline{u}$ be the element selected from $U_{d'\underline{d}}$. Delete the element from $U_{d'\underline{d}}$.

Step 8. Search in R for an element (u'$\underline{u}$,'x','x'). If (u'$\underline{u}$, 'x', 'x') $\in$ R, then go to Step 10.

Step 9. Add (u'$\underline{u}$, $\emptyset$, {d'$\underline{d}$}) to R. Also, add u'$\underline{u}$ to $DB_{d'\underline{d}}$. Go to Step 11.

Step 10. If $DB_{u'\underline{u}} = \emptyset$, then add d'$\underline{d}$ to $UB_{u'\underline{u}}$ and also add u'$\underline{u}$ to $DB_{d'\underline{d}}$; otherwise, for each element y'$\underline{y}$ in the set $DB_{u'\underline{u}}$, add y'$\underline{y}$ to $DB_{d'\underline{d}}$, and further if $\underline{y} \neq \underline{c}$, add d'$\underline{d}$ to $UB_{y'\underline{y}}$.

Step 11. Go to Step 7.

Theorem 3. Algorithm 4 identifies the intrablock error flow.

Proof. Given the set DU constructed for a statement contained in a block v_i, and the partially constructed set R for v_i, we would like to show that Algorithm 4 correctly identifies the intrablock error flow in v_i up to the end of the statement.

The algorithm processes the elements in DU sequentially. For each element (d'$\underline{d}$, $U_{d'\underline{d}}$) in DU, the definition d'$\underline{d}$ is processed first. Then, each element in $U_{d'\underline{d}}$ is processed one by one. Thus, it is sufficient to show that the algorithm correctly identifies the intrablock error flow in v_i after each definition and all the usages used to define the definition have been processed.

Let (d'$\underline{d}$, $U_{d'\underline{d}}$) be the element selected from DU at one stage of execution. To process the definition d'$\underline{d}$, the usage set associated with d'$\underline{d}$ will be searched first to determine if d'$\underline{d} \in U_{d'\underline{d}}$. If d'$\underline{d} \in U_{d'\underline{d}}$, i.e., d'$\underline{d}$ is currently defined by a usage of itself, the flag f_1 is set to be 'true'. Otherwise, f_1 is set to be 'false'. Then, the set R is searched to determine if (d'$\underline{d}$, 'x', 'x') is contained in R. The following two cases have to be considered.

Case 1. ($d'\underline{d}$, 'x', 'x') $\notin$ R, i.e., the definition to $d'\underline{d}$ is the first reference to $d'\underline{d}$ in v_i. In this case, if f_1 = 'true', we should add ($d'\underline{d}$, $\{d'\underline{d}\}$, $\{d'\underline{d}\}$) to R because $d'\underline{d}$ can propagate potential errors to itself due to this definition by a usage of itself. If f_1 = 'false', we should add ($\{d'\underline{d}\}$, $\emptyset$, $\emptyset$) to R because $d'\underline{d}$ can not propagate potential errors to any items looking at this point, and the set of items which can propagate potential errors to $d'\underline{d}$ due to this definition will be identified when $U_{d'\underline{d}}$ is processed later. Case 1 is handled by the Step 4 in the algorithm.

Case 2. ($d'\underline{d}$, 'x', 'x') $\in$ R, i.e., the item $d'\underline{d}$ has been previously referenced in v_i. In this case, if $DB_{d'\underline{d}} = \emptyset$, i.e. $d'\underline{d}$ has not been previously defined in v_i, and f_1 = 'true', we should add $d'\underline{d}$ to both $DB_{d'\underline{d}}$ and $UB_{d'\underline{d}}$, since $d'\underline{d}$ can propagate potential errors to itself due to this definition by a usage of itself. Now, if $DB_{d'\underline{d}} \neq \emptyset$, i.e., $d'\underline{d}$ has been previously defined in v_i, and f_1 = 'false', then the elements in $DB_{d'\underline{d}}$ can no longer propagate potential errors to $d'\underline{d}$ due to this redefinition of $d'\underline{d}$. Hence, for each element $y'\underline{y}$ in $DB_{d'\underline{d}}$ and $\underline{y} \neq \underline{c}$, we should delete $y'\underline{y}$ from $DB_{d'\underline{d}}$ and also delete $d'\underline{d}$ from $UB_{y'\underline{y}}$. For the other two combinations, i.e., either $DB_{d'\underline{d}} = \emptyset$ and f_1 = 'false', or $DB_{d'\underline{d}} \neq \emptyset$ and f_1 = 'true', nothing has to be done at this point, because $d'\underline{d}$ can still propagate potential errors to the elements in $UB_{y'\underline{y}}$ through previous usages of $d'\underline{d}$, and the elements in $DB_{d'\underline{d}}$ can still propagate potential errors to $d'\underline{d}$. Case 2 is handled by the Step 5 in the algorithm.

For both cases, if $U_{d'\underline{d}} = \emptyset$, i.e., $d'\underline{d}$ is defined by an expression in which only constants and/or operators are involved, we should add $c'\underline{c}$ to $DB_{d'\underline{d}}$ in order to denote that $d'\underline{d}$ is defined by an expression which evaluates a constant value, and branch to process the next element in R. If f_1 = 'true', we should delete $d'\underline{d}$ from $U_{d'\underline{d}}$, since both the definition and usage of $d'\underline{d}$ have been processed. Step 6 in the algorithm describes this operation, and Steps 2-6 process a definition $d'\underline{d}$.

Now, for each element $u'\underline{u}$ in $U_{d'\underline{d}}$, we know that $u'\underline{u} \neq d'\underline{d}$, since $d'\underline{d}$ has been deleted from $U_{d'\underline{d}}$ if it was originally contained in $U_{d'\underline{d}}$. If ($u'\underline{u}$, 'x', 'x') $\notin$ R, then we should add ($u'\underline{u}$, $\emptyset$, $\{d'\underline{d}\}$) to R, since $u'\underline{u}$ is currently used to define $d'\underline{d}$ and hence $u'\underline{u}$ can propagate potential errors to $d'\underline{d}$. The Step 9 in the algorithm handles this case. If ($u'\underline{u}$, 'x', 'x') $\in$ R, then we have to

check if $DB_{u'\underline{u}} = \emptyset$. If $DB_{u'\underline{u}} = \emptyset$, i.e., $u'\underline{u}$ has not been previously defined in v_i, then we should add $u'\underline{u}$ into $DB_{d'\underline{d}}$ and also add $d'\underline{d}$ into $UB_{u'\underline{u}}$, since $u'\underline{u}$ can propagate potential errors to $d'\underline{d}$. If $DB_{u'\underline{u}} \neq \emptyset$, the elements in $DB_{u'\underline{u}}$ which can propagate potential errors to $u'\underline{u}$ can also propagate potential errors to $d'\underline{d}$ through the usage of $u'\underline{u}$ in defining $d'\underline{d}$. Hence, for each element $y'\underline{y}$ in $DB_{u'\underline{u}}$, we should add $y'\underline{y}$ to $DB_{d'\underline{d}}$; and if $\underline{y} \neq \underline{c}$, we should add $d'\underline{d}$ to $UB_{y'\underline{y}}$. Note that we do not have to add $d'\underline{d}$ to $UB_{u'\underline{u}}$, since the usage of $u'\underline{u}$ appears after $u'\underline{u}$ has been defined in v_i and hence $u'\underline{u}$ cannot propagate potential errors from outside of v_i to $d'\underline{d}$ except when $u'\underline{u}$ can propagate potential errors from outside of v_i to $u'\underline{u}$. In that case, $u'\underline{u}$ should have already been in $UB_{u'\underline{u}}$, and hence $d'\underline{d}$ should have been added to $UB_{u'\underline{u}}$ when we add $d'\underline{d}$ to $UB_{y'\underline{y}}$, for each $y'\underline{y}$ in $UB_{u'\underline{u}}$ such that $\underline{y} \neq \underline{c}$. Step 10 in the algorithm handles the case that $(u'\underline{u}, 'x', 'x') \in R$. Steps 7 to 11 process a usage $u'\underline{u}$ which is used to define $d'\underline{d}$.

Thus, we have shown that the execution of Steps 2-6 and the iterations of Steps 7-11 (each iteration for a usage used to define the definition) correctly identifies the intrablock error flow based on the set R, the definition and its associated usage set. This proves that Algorithm 4 correctly identifies the intrablock error flow based on the sets R and DU.

To show that Algorithm 4 terminates, we observe that each execution of Step 1 decrements the cardinality of DU by one. To process an element selected from DU, Steps 2-6 will be executed only once to process the definition. Each iteration of Steps 7-11 will decrement the cardinality of the usage set associated with the definition. Since each usage set is of finite size, it takes a finite number of iterations to process any usage set. Besides, it takes a finite number of steps to do searching or checking. Hence, it takes a finite number of steps to process an element in DU, and to process the whole set DU. Thus, Algorithm 4 terminates. This completes the proof of the theorem.

Upon exiting from a block v_i, the reference set identified for the block is processed by the block error characteristics identification algorithm to derive the error characteristics of the block. Given the reference set R identified in v_i, and the two sets P_i and C_i which were initialized to be empty upon entering the block, the block error characteristics identification

algorithm can be expressed as follows:

Algorithm 5. Block Error Characteristics Identification Algorithm

Step 1. Select an element ($r'\underline{r}$, $DB_{r'\underline{r}}$, $UB_{r'\underline{r}}$) from the set R. If R is empty, then terminate. Delete the element from R.

Step 2. If $DB_{r'\underline{r}}$ is not empty, then add r into C_i. If $UB_{r'\underline{r}}$ is not empty, then add r into P_i and let r map to the set $\{y\}$ through the flow mapping f(r), where $y'\underline{y}$ is an element in $UB_{r'\underline{r}}$.

It is obvious that Algorithm 5 terminates. For each element ($r'\underline{r}$, $DB_{r'\underline{r}}$, $UB_{r'\underline{r}}$) selected from R, if $DB_{r'\underline{r}}$ is not empty, i.e., $r'\underline{r}$ has been defined in v_i, then r should be added to C_i, since it can cause potential errors to exist in v_i. Furthermore, if $UB_{r'\underline{r}}$ is not empty, i.e., $r'\underline{r}$ can propagate potential errors to the elements in $UB_{r'\underline{r}}$ from outside of v_i, then r should be added to P_i and it should map to the set $\{y\}$ through the flow mapping $\bar{f}(r)$, where $y'\underline{y}$ is an element in $UB_{r'\underline{r}}$. Thus, we conclude that Algorithm 5 terminates and correctly identifies the error characteristics of the block v_i based on the set R constructed for v_i.

3.2.2 *Lexical Analysis Step Two*

The main function of this step of the lexical analysis phase is to utilize the output of Step One in order to compute the error characteristics of the modules in the program, and thus to characterize the potential error properties of the program.

In this phase, the reverse invocation order is derived from the program call graph which was constructed in Step one of lexical analysis. Then, the error characteristics for all modules except the initial entry module are defined following the reverse invocation order. After the error characteristics of a module M_j are defined, the error characteristics of the blocks in other modules which are assigned for invocations to M_j are then updated. Recall that a sequence of one to three blocks may be assigned for each module invocation. The module error characteristics of M_j are then assigned to the block error characteristics of the module invocation block in M_k assigned for an invocation to M_j, where $M_j \in I(M_k)$. The formal parameter list of M_j and the actual parameter list in the module invocation are then scanned to derive the block error characteristics of the input parameter error flow mapping block. The definition-usage set is first constructed by treating each actual

input parameter as a usage which is used to define its corresponding formal input parameter. Then, the intrablock error flow identification algorithm is invoked to derive the set R from the set DU. Finally, the block error characteristics of the input parameter error flow mapping block can be derived from the set R by the block error characteristics identification algorithm. The block error characteristics of the output parameter error flow mapping block can also be derived in the same manner except that the definition-usage set is constructed by treating each formal output parameter as a usage which is used to define its corresponding actual output parameter.

3.2.3 Tracing Phase

In this section, the tracing phase of the logical ripple effect analysis technique will be described. Upon completion of lexical analysis of the modified program, this phase is entered to compute the ripple effect caused by the initial modification. First, the primary error sources in the blocks and modules which are involved in the initial modification must be identified. Then, Algorithm 3 is applied to trace the error flow in the program utilizing the primary error sources as starting points. Finally, a logical ripple effect criterion is used to identify the logical ripple effect based on the output of Algorithm 3. In this section, the logical ripple effect identification algorithm and a process for identifying primary error sources will be presented.

Recall that Algorithm 3 determines which definitions, blocks, and modules are logically affected by the propagation of potential errors. However, Algorithm 3 does not determine if the definition, block, or module actually requires additional maintenance. For example, a module M_j requires no additional maintenance despite the fact that it is affected by intermodule error flow if all the error sources identified in M_j are just passing through M_j to the modules invoked by M_j without propagating potential errors within M_j.

A criterion is thus needed to distinguish between the modules which are actually affected by logical ripple effect and those which are not. Let G_j' be a subgraph of G_j which is derived by deleting from V_j all the blocks in M_j which are assigned for module invocations in M_j. Informally, a module M_j is affected by logical ripple effect if there exists at least one block in G_j' which is affected by error flow. The modules in a program which are affected

by logical ripple effect can then be identified based on the following lemma:

Lemma 4. (Logical Ripple Effect Criterion). Given the propagation error source set $\mathcal{S}_i$ of each block v_i in a module M_j, the module M_j is affected by logical ripple effect from the initial modification if there exists a block $v_k \in G'_j$ such that $(\mathcal{S}_k \cap C_k) \neq \emptyset$.

Proof. $\mathcal{S}_k \neq \emptyset$ implies that M_j is suspected of being affected by logical ripple effect from the initial modification. $\mathcal{S}_k \cap C_k = \emptyset$ infers that v_k is unaffected by the logical ripple effect, but does not imply that M_j is also unaffected. However, $\mathcal{S}_k \cap C_k = \emptyset$ for each block v_k in G'_j implies that all the blocks in M_j which are not assigned for module invocations are not affected by the error flow. Thus, M_j is not affected by the logical ripple effect because all the error sources which propagate in M_j are just passing through M_j to the modules invoked by M_j. Hence, the criterion that there exists a $v_k \in G'_j$ such that $\mathcal{S}_k \cap C_k \neq \emptyset$ is sufficient to determine that M_j is affected by the logical ripple effect of the initial modification.

In an analogous manner, a block may require no additional maintenance despite the fact that it is affected by the error flow. Thus, a criterion is needed to eliminate the blocks which are not affected by the error flow.

Lemma 5. (Block Elimination Criterion). If a module M_k is identified by the logical ripple effect criterion as not being affected by the logical ripple effect, then all the blocks in M_k require no additional maintenance. Furthermore, if $M_j \in I^{-1}(M_k)$, and $(v_i, M_k, \overline{P}_{v_i}) \in \mathcal{J}_j$, where v_i is the module invocation block assigned in M_j for an invocation of M_k, then the block v_i and the blocks in $\overline{P}_{v_i}$ require no additional maintenance.

Proof. The fact that M_k is identified as not being affected by the logical ripple effect implies that M_k does not contain any blocks affected by the intermodule error flow that would disturb its consistency. Hence, all the blocks in M_k require no additional maintenance.

If M_j invokes M_k and v_i is a module invocation block assigned for an invocation of M_k in M_j, then the block v_i and the blocks in $\overline{P}_{v_i}$ which are assigned for the invocation require no additional maintenance because the error flow does not disturb M_k's consistency, and hence the consistency of these blocks is not disturbed.

Let $\mathcal{M}^R$ denote the set of modules which are affected by the logical ripple effect. For each module M_j in $\mathcal{M}^R$, let the set L_j^R consist of the blocks in M_j which are affected by the logical ripple effect and their associated error sources. Given the set $\mathcal{M}^*$ of the modules affected by the error flow and the sets L_k' of the modules in $\mathcal{M}^*$ identified by Algorithm 3, the logical ripple effect identification algorithm which derives the set $\mathcal{M}^R$ and the sets L_j^R of the modules in $\mathcal{M}^R$ can be presented as follows:

Algorithm 6. Logical Ripple Effect Identification Algorithm

Step 1. Initialize the set $\mathcal{M}^R$ to be empty. Apply the logical ripple effect criterion to each module in $\mathcal{M}^*$. If a module is identified as affected by the logical ripple effect, then add it into $\mathcal{M}^R$. After all the modules in $\mathcal{M}^*$ have been examined, the set $\mathcal{M}^R$ contains all the modules affected by the logical ripple effect.

Step 2. Calculate the set of modules which are affected only the error flow, but not by the logical ripple effect as $\mathcal{M}^* - \mathcal{M}^P$. For each module $\mathcal{M}_j$ in $\mathcal{M}^R$, assign L_j' to L_j^R. For each module M_k in $\mathcal{M}^* - \mathcal{M}^R$, delete the blocks with their associated error sources which are assigned for invocations to M_k from the respective L_j^R, where M_j invokes M_k and is a member of $\mathcal{M}^R$.

It is clear from Lemmas 4 and 5 that Algorithm 6 correctly computes the logical ripple effect based on the intermodule error flow identified in the program.

Thus, maintenance personnel should check the blocks and their error sources in the L_j^R sets of the modules in $\mathcal{M}^R$ to insure their logical consistency with the initial modification.

Before the logical ripple effect identification algorithm can be applied, it is first necessary to identify all of the primary error sources involved in the initial modification. These primary error sources serve as the starting points to trace the error flow in the program. A primary error source can be defined as a data or control definition which is directly affected or implicated by the initial modification. A directly affected primary error source is a definition whose value or control condition associated with it was directly changed by the initial modification. Implicated primary error sources are the data or control items which are defined with direct or indirect usages of some directly affected primary error sources of the block.

Implicated primary error sources are required because the ripple effect analysis technique starts tracing logical ripple effect from the successor blocks of the blocks which are involved in the initial modification. Hence, the maintenance programmer must identify the definitions affected by the intra-block error flow within the primary error source blocks. If a definition employing explicit addressing is identified as a primary error source and is later redefined in the block without usages of any affected data items, then the definition can no longer propagate potential errors to other blocks, and hence must be removed from the set of primary error sources of the block.

Another type of complication arises when the control flow is changed due to the deletion of some code. Since the ripple effect analysis technique is based on the potential error properties of the modified program, some potential errors may not be traceable due to the change in control flow. In order to solve this problem, the maintenance personnel must specify the deleted definitions as directly affected primary error sources for the blocks in the modified program to which the deleted code could transfer control flow. These blocks of the modified program should then be specified as primary error source blocks.

Based on the above discussion, the emphasis here will be on how to identify the directly affected primary error sources due to a program modification To illustrate this, let us consider the following modifications:

- Suppose that the data items used to define a control condition were changed, e.g. a loop termination condition was modified. The control definition associated with this control condition is then specified as a directly affected primary error source of the block, to which the control definition is assigned.
- Suppose that a data definition was changed, added, or deleted in a block. The definition is then specified as a directly affected primary error source of the block.
- Suppose that parameter x was replaced by y in a module invocation. If the corresponding formal parameter f is an input parameter, then f is specified as a primary error source of the input parameter error flow mapping block associated with this module invocation. If f is an output parameter, then x and y are both specified as primary error sources of the output parameter

error flow mapping block.

- Suppose that a module invocation which invokes a newly added or an existing module was inserted into the program. The invoked module's natural source capable definitions are then specified as primary error sources of the module invocation block associated with this newly added module invocation.
- Suppose that a module invocation which invokes M_k was deleted from a module M_j. The directly affected primary error sources are then M_k's natural source capable definitions, except that the formal output parameters should be replaced by their corresponding actual output parameters which has appeared in the deleted module invocation.

Utilizing these guidelines, the set $\bar{m}$ of modules which are involved in the initial modification and the sets $\mathcal{A}_j$'s which consist of the blocks and their associated primary error sources in the modules in $\bar{m}$ can then be defined by the maintenance personnel and input to the intermodule error flow tracing algorithm.

3.3 Implementation Considerations for a Restricted JOVIAL Language

The logical ripple effect analysis models and the logical ripple effect analysis techniques previously described are intended to be language-independent and system-independent. We have left out the descriptions of some important functions which must be performed in the lexical analysis phase due to their language dependence. In this section, we will describe the overall structure of a technique which aims at performing the logical ripple effect analysis on programs written in a restricted JOVIAL language. We will also describe a scheme which derives the program graphs associated with program modules, and a scheme which identifies definitions and usages from various syntactical constructs allowed in our restricted JOVIAL language.

3.3.1 Restricted JOVIAL

Our restricted JOVIAL retains the basic and most desirable features of the JOCIT JOVIAL. Programs written in our restricted JOVIAL can ultimately carry out all the functions supported by JOCIT JOVIAL. Our restricted JOVIAL imposes some limitations on the execution control transfer mechanisms allowed in JOCIT JOVIAL. These limitations are described as follows:

1) Statement names and module names cannot be passed between modules. A module is defined to be a separately invokable piece of the software

system having a single entry point and a single exit points. If statement names and modules names can be passed between modules, then the invoked module may have multiple entry or multiple exit points. However, current programming practices try to avoid this feature because it leads to bad programming style. Thus, it can be justified that statement names and module names should not be passed between modules.

2) Switches which can cause execution control to be transferred depending on the values of data items or index expressions are not allowed in our restricted JOVIAL since they can also complicate the execution control and lead to bad programming style.
3) The TEST statements which provide a special loop exit mechanism are also not allowed in our restricted JOVIAL.

3.3.2 System Structure

The system to perform the logical ripple effect analysis on restricted JOVIAL programs can be decomposed into three automated subsystems: a Text Analyzer, a System-Level Lexical Analyzer, and a Logical Ripple Effect Calculator. A logical ripple effect analysis data base is utilized to maintain all the information required for logical ripple effect analysis.

A text is defined to be an entity which is compiled independently. In JOVIAL, a text can be a compool, a main program or a subprogram; i.e. any START-TERM sequence. Each text in the modified program must be processed by the Text Analyzer in the same order as defined by the compilation process of the modified program, i.e. compools must be processed prior to the main and subprograms.

The main functions performed by the Text Analyzer are:

1) Resolve address conflicts and establish symbol tables.
2) Identify the global and passed parameter sets of the modules defined in the text.
3) Identify the blocks with their respective block error characteristics in each module defined in the text.
4) Derive the program graph associated with each module defined in the text.
5) Identify the immediate successor and predecessor modules of each module defined in the text.

The Text Analyzer can be devised as a two-pass processor to lexically analyze a text. The Pass One of the Text Analyzer will perform the functions 1) and 2). The Pass Two then performs the functions 3), 4) and 5). Hence, correct block error characteristics can be identified since address conflicts have been resolved first.

A reformed source text output will be produced by the Text Analyzer after a text has been processed. The reformed source text output is basically the card image of the source text input except that all address conflicts have been resolved. Hence, some data item names may be relabelled. Furthermore, some block indicators which denote the boundaries of the blocks, and control definition indicators which denote the control items assigned for conditional or FOR clauses may be inserted into the reformed source text output for a main or subprogram which contains executable statements. These indicators are provided to the maintenance personnel for the purpose of identifying primary error sources involved in the initial modification to the text.

The functional step which performs the text-level lexical analysis on each text in the modified program can be illustrated by Figure 5.

After all the texts in the modified program have been processed by the Text Analyzer, the System-Level Lexical Analyzer is invoked to perform the system-level lexical analysis. The main functions performed by the System-Level Lexical Analyzer are:

1) Derive the reverse invocation order based on the call graph which was constructed during the text-level lexical analysis and represented by the predecessor-successor relationships among modules.
2) Derive the module error characteristics for each module (except the main module) in the modified program following the reverse invocation order obtained beforehand.
3) For each module invocation block, update its error characteristics which were unspecified during the text-level lexical analysis by the module error characteristics of the invoked module.

The System-Level Lexical Analyzer performs the functions mentioned above based on the information contained in the Logical Ripple Effect Analysis Data Base. A report which indicates the natural source capable definitions identified for respective modules is produced by the analyzer to aid the

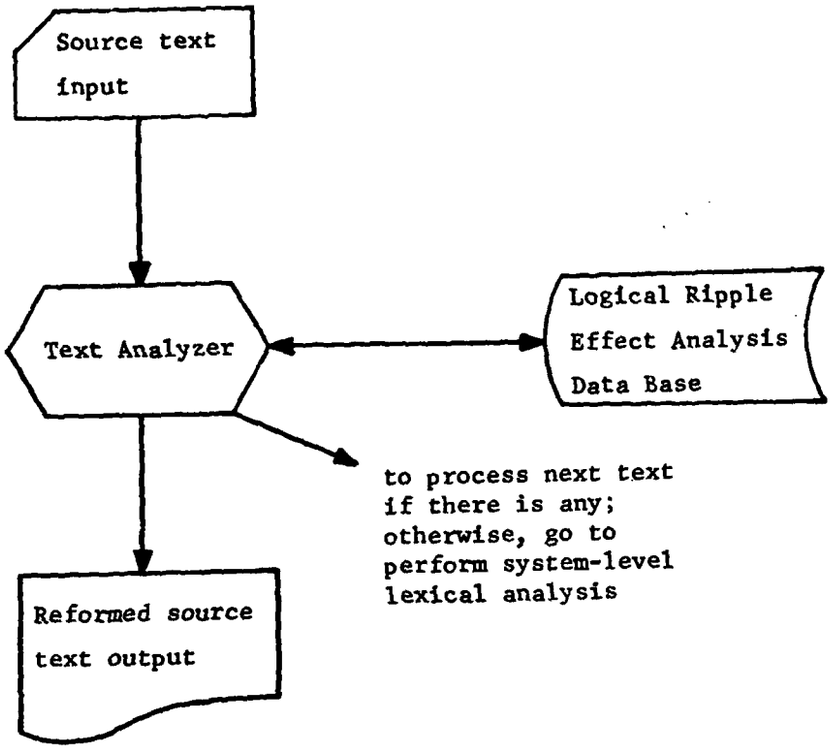


Figure 5. The block diagram of the text-level lexical analysis step.

maintenance personnel in identifying the primary error sources involved in initial modifications to module invocations. The system-level lexical analysis functional step can be illustrated by Figure 6.

After the system-level lexical analysis has been performed, the characterization of the error properties of the program is completed. The maintenance personnel then follow the guidelines as described in Section 3.2.3 to identify the primary error sources utilizing the reformed source text outputs and the natural source capable definition report. Then, the Logical Ripple Effect Calculator is invoked to compute the logical ripple effect based on the primary error sources supplied by maintenance personnel and the error characterization of the modified program obtained in the lexical analysis phase. A logical ripple effect report will be produced by the Logical Ripple Effect Calculator to indicate the definitions, blocks, and modules which are

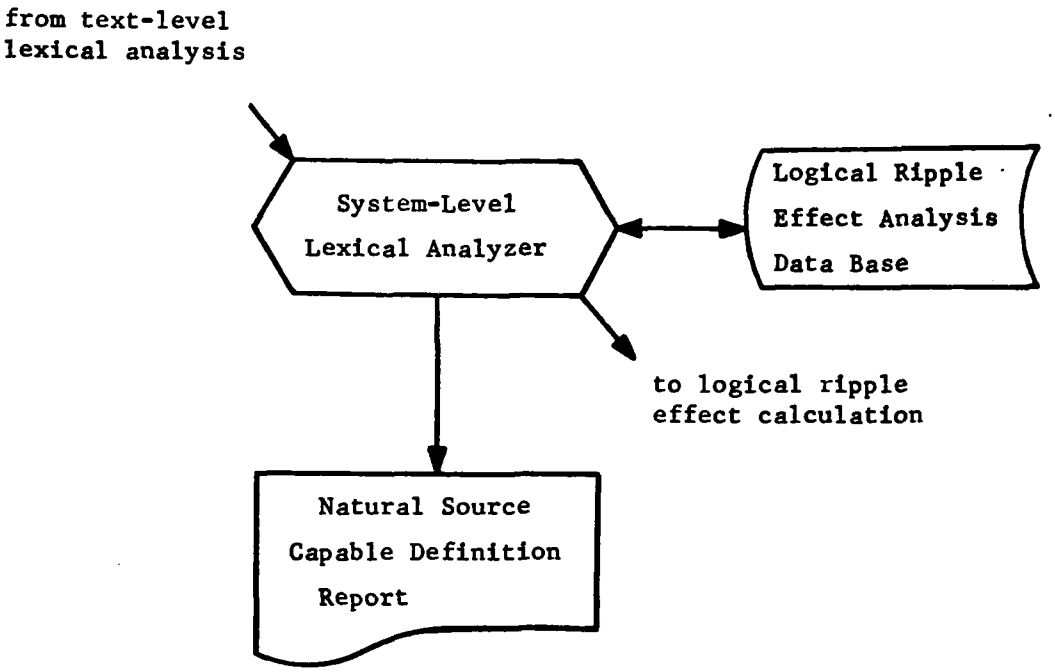


Figure 6. System-level lexical analysis step.

affected by the logical ripple effect. The logical ripple effect calculation functional step can be illustrated by Figure 7.

3.3.3 Some Important Aspects of the Text Analyzer

As we described in Section 3.3.2, the Text Analyzer performs the text-level lexical analysis. The most challenging functions performed by the Text Analyzer are: to identify the blocks with their respective error characteristics in each module, and to derive the program graph associated with each module defined in the text being processed. In this section, we will present the schemes which can be utilized to derive the program graph associated with a module, and to identify the definitions and usages from various syntactical constructs in our restricted JOVIAL.

3.3.3.1 Derivation of the Program Graph

A program block is defined to be the maximal sequence of computer statements having the property that each time any statement in the sequence is executed, all are executed; except that when a module invocation is

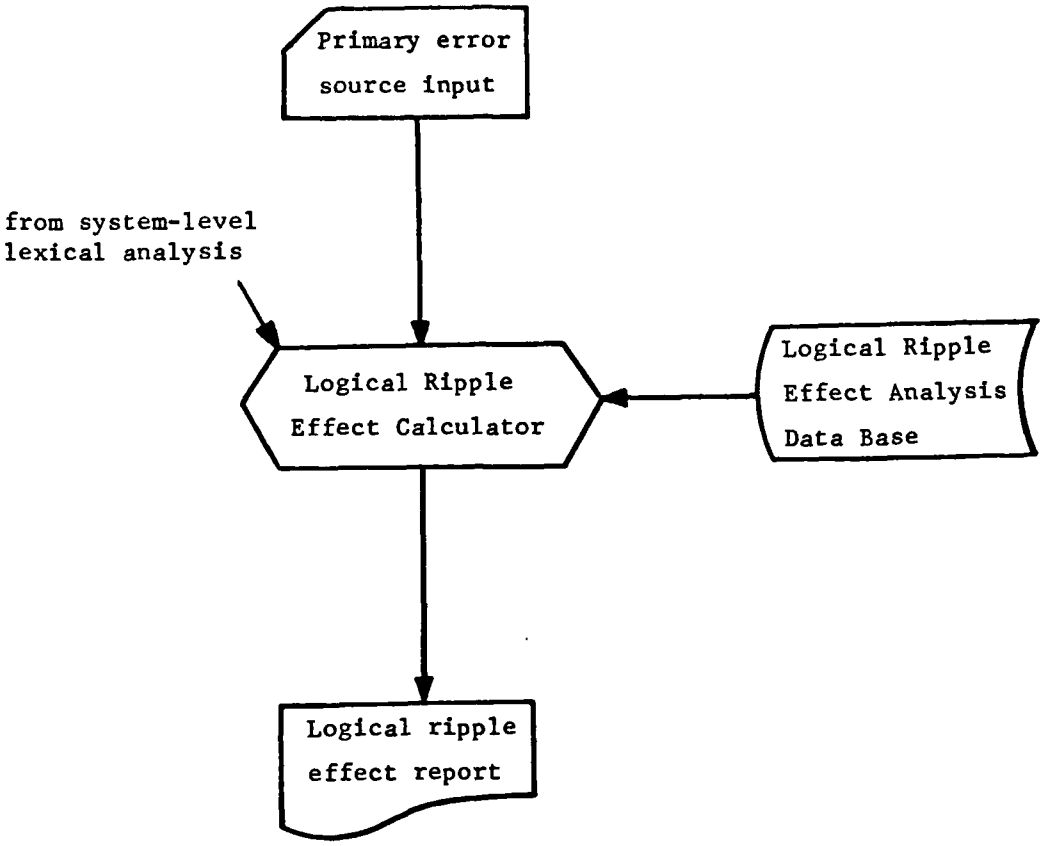


Figure 7. Logical ripple effect calculation step.

encountered, a sequence of one to three blocks may be assigned for this invocation. However, some variations on the maximal condition are made to facilitate implementing this technique. A program graph associated with a module can be represented by the immediate predecessor-successor relationships among the blocks in the module.

The scheme to derive the program graph can be described by a set of block identification conditions which provides a criterion on how a program module should be partitioned into program blocks. These block identification conditions should identify all control flow changes other than those in sequential control flow as well as other special conditions which facilitate the tracing of the error flow.

For our restricted JOVIAL, nineteen block identification conditions have

been identified. In the following description, sequential control flow among blocks is assumed unless it is stated otherwise. The block identification conditions are described as follows:

1) The module declaration header constitutes the entry block of the module. The first block in the module is identified upon entering the body of the module.
2) After reaching the end of the module body, a new block is identified as the exit block of the module. The exit block contains no executable statements. However, it is required for intermodule error flow tracing.
3) A GOTO statement which references a statement name should be the last statement in a block. The block which contains the statement bearing the name referred by the GOTO statement is the immediate successor of the block containing the GOTO statement.
4) A RETURN statement should be the last statement in a block. The exit block of the module is the immediate successor of the block which contains the RETURN statement.
5) A STOP statement should be the last statement in a block. If the module is an external close, the exit block of the module is the immediate successor of the block containing the STOP statement; otherwise, the block which contains the STOP statement has no immediate successor.
6) A statement bearing a name should be the first statement in a block.
7) An input statement should be the first statement in a block. This condition is required to facilitate tracing the error flow since an inconsistent input file can propagate potential errors to the data items in the input list.
8) An output statement should be the last statement in a block. This condition is also required to facilitate tracing the error flow since the data items in the output list can cause the output file to be inconsistent.
9) A FOR clause should constitute a block by itself.
10) The end of the iteratively executable simple or compound statement associated with a FOR clause should be the end of a block. The block ended by this condition has two immediate successors: the block which contains the corresponding FOR clause, and the block which follows the end of the FOR statement. Combining the Conditions 9) and 10), the control flow

among the blccks identified for a FOR statement can be illustrated by Figure 8.

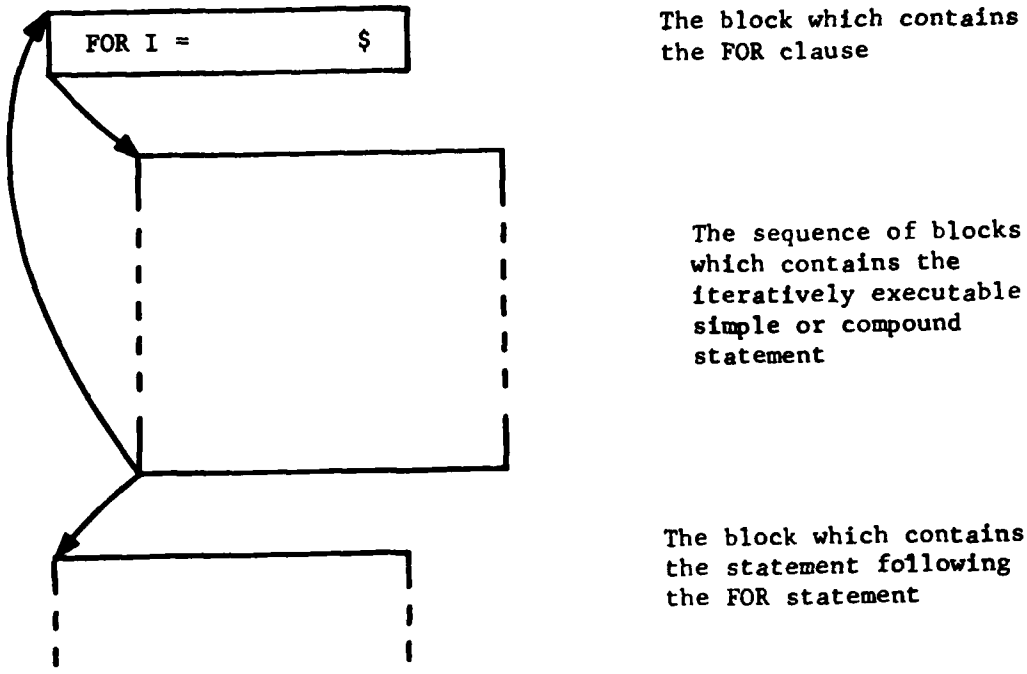


Figure 8. An illustration for block segmentation for a FOR statement

11) An IF clause should be the end of a block. The block ended by the IF clause has two immediate successors: the first block identified for the conditionally executable statement associated with the IF clause, and the block which follows the end of the IF statement.

12) The end of the conditionally executable statement associated with an IF clause should be the end of a block. If the block is also ended by Condition 3), 4) or 5), then the block cannot transfer control to the block following the end of the IF statement. Combining the Conditions 11) and 12), the control flow among the blocks identified for an IF statement can be illustrated by Figure 9.

13) An IFEITH clause should be the end of a block. The block ended by this condition has two immediate successors: the first block identified for

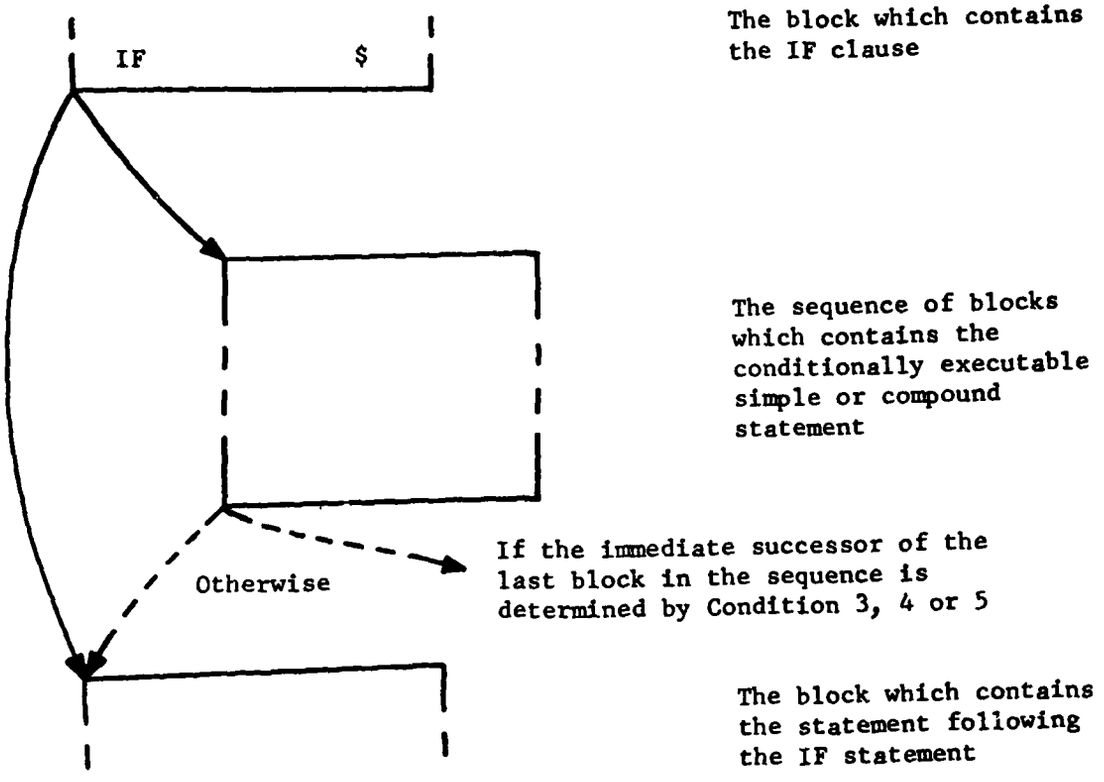


Figure 9. An illustration for block segmentation for an IF statement.

the conditionally executable statement associated with the IFEITH clause, and the block which contains the first ORIF clause in this IFEITH-ORIF's complex construct.

14) An ORIF clause should constitute a block by itself. This block has two immediate successors: one is the first block identified for the conditionally executable statement associated with the ORIF clause, and the other is the block which contains the next ORIF clause in the IFEITH-ORIF's complex construct if there is one, or the block which contains the end of the construct.

15) The end of the conditionally executable statement associated with an IFEITH or ORIF clause should be the end of a block. If the block is also ended by Condition 3), 4) or 5), then the immediate successor of the

block is determined by that condition. Otherwise, the block which contains the end of the IFEITH-ORIF's construct is the immediate successor of this block.

16) The END bracket, which denotes the end of the IFEITH-ORIF's construct, should be the first statement in a block. Combining the Conditions 13) to 16), the control flow among the blocks identified for an IFEITH-ORIF's construct can be illustrated by Figure 10.

17) A GOTO statement which invokes a close should constitute a block by itself. The block is the module invocation block for this close invocation.

18) A block is identified for a function reference without passed parameters. The block is the module invocation block. For a function reference with passed parameters, a sequence of two blocks is identified for the function reference, where the first block is the input parameter error flow mapping block, and the second block is the module invocation block. The immediate successor of the module invocation block, for both cases, is the block which contains the statement involving the function reference.

19) A block is identified for a procedure call statement which involves no passed parameters. The block is the module invocation block. A sequence of three blocks is identified for a procedure call statement if it involves passed parameters.

The set of block identification conditions forms a criterion for deriving the program graph for our restricted JOVIAL.

3.3.3.2 Identification of Definitions and Usages

The error characteristics of a block are derived from the definitions, usages and their relationships identified in the block. The scheme which identifies definitions and usages from various syntactical constructs of a programming languages is essential in correctly identifying block error characteristics.

The scheme is described here by a set of rules which provide the criterion on which data or control items should be identified as definitions or usages from various syntactical constructs. Before we state the rules for our restricted JOVIAL language, we emphasize two rules which are general to

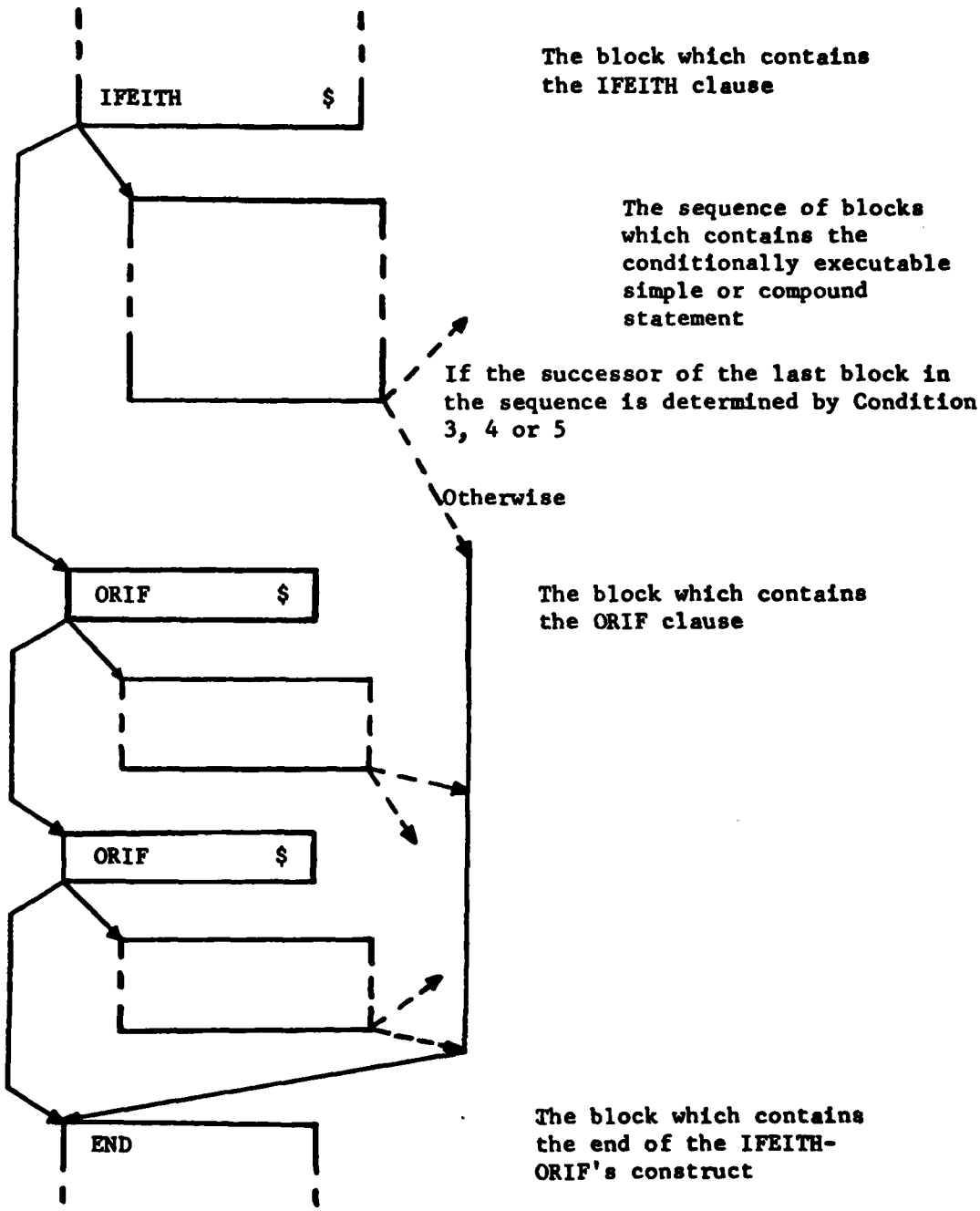


Figure 10. An illustration for block segmentation for an IFEITH-ORIF's construct.

most programming languages.

1) When a loop control variable is referenced within the scope of the loop, the control item which is assigned to represent the control directive of the loop execution condition, instead of the loop control variable, is treated as a usage used to define the relevant definition(s).
2) The data or control items which are used to dynamically evaluate the address of the referenced element in a data structure are always treated as usages.

For our restricted JOVIAL language, except the exchange statements, the rules which determine which data or control items should be identified as definitions or usages are stated as follows:

1) In general, the first data item which appears in an assignment statement is the definition while other data or control items appearing in the statement are usages used to define the definition, except when a functional modifier variable (BIT or BYTE) is assigned new value by the assignment statement. In that case, the first data item enclosed by the parentheses in defining the functional modifier variable is the definition while other data or control items appearing in the statement are usages used to define the definition.
2) In an input statement, the input file is a usage which is used to define all the data items which appear in the input list, except those which also appear in some index list. A data or control item which appears in the index list when an element in a data structure is referenced is a usage used to define the referenced compound data item.
3) In an output statement, the output file is a definition, while the data or control items appearing in the output list are usages used to define the definition.
4) In an IF, IFEITH, ORIF, or FOR clause, the control item assigned to represent the control directive of the clause is the definition while the data or control items appearing in the control directive are usages used to define the definition.
5) In an ENCODE or DECODE statement, the data items appearing on the right hand side of the equal sign are definitions, except those which also appear in some index list. In that case, the item appearing in an index

list is a usage used to define the referenced compound data item. Furthermore, the data or control items which appear on the left hand side of the equal sign are usages used to define all the definitions.

6) For a module invocation with passed input parameters, the formal input parameters are definitions, while the data or control items appearing in the actual input parameter list are usages used to define their corresponding formal input parameters.

7) For a module invocation with passed output parameters, the formal output parameters are usages used to define their corresponding actual output parameters, except those actual output parameters which also appear in some index list. In that case, the item appearing in an index list is a usage used to define the referenced compound data item.

8) For a function reference to any intrinsic function allowed in JOVIAL, the data or control items appearing in the actual parameter list are usages used to define the relevant definitions which are determined by the statement or expression containing the function reference.

9) For a procedure call statement which invokes the REMQUO intrinsic procedure in JOVIAL, all the data or control items which appear in the actual input parameter list are usages used to define the data items which appear in the actual output parameter list, except those actual output parameters which also appear in some index list. In that case, the data or control item which appears in an index list is a usage used to define the referenced compound data item.

The rules described above provide the criteria on which data or control items should be identified as definitions or usages from various syntactical constructs in our restricted JOVIAL.

3.4 Conclusion

A logical ripple effect analysis technique based upon intramodule and intermodule error flow models has been developed, and the implementation of this technique also discussed. The importance of this technique lies in its capability to trace the logical ripple effect in a program as a consequence of a modification, and thus help maintenance personnel better understand the scope of the ripple effect of their changes on the program.

Much work still needs to be done in this area. Our immediate objective is to expand the logical ripple effect analysis technique to handle recursion. The technique will also be improved in order to refine its current worst case logical ripple effect analysis, and to enhance the efficiency and capacity for full automation of the technique.

4.0 GENERATION OF SPECIFICATION FOR PROGRAM MODIFICATIONS

This part of the work is to describe a given program modification for an existing program. This description must be unambiguous, easy to understand, and easy to construct. All of this implies that we must have a clear and consistent way of describing a program. Therefore, we need a representation of the program that identifies all of the program elements and the relations among those elements in a well-defined way. This should not only allow a clear description of proposed modifications to the existing program, but also aid the programmer in understanding the function and structure of the program. It is our intention to formally describe a given program modification based on a formal program model. Here, we outline some program abstractions (e.g. control flow, data flow, execution flow, and data object structure) and some operations on the abstractions which could be used to formally describe a given program modification. In order to illustrate our approach, we use the abstractions to describe two FORTRAN subroutines. We also describe how our approach can be used to specify a modification to one of the subroutines.

To date, our approach has not been formalized. We only present an outline and some informal definitions here. The four abstractions which we will describe have neither been formalized, nor incorporated into a unified model. During the development of our research, we have come to the conclusion that a formal and unified model of an existing program is necessary for the formal specification of a given program modification. Further research needs to be done to formally define such a model.

4.1 Some Abstractions

We define an abstraction as a representation of the inherent properties of something from the actual object to which they belong. The "object" to be described is the existing program and any abstraction of the program will be a representation of some aspect of the program.

There are four aspects of a program which we identify: control flow, data flow, execution flow, and data object structure. In the following sections, we will discuss the construction of an abstraction for each of these four aspects.

4.1.1 Previous Uses of Program Abstractions

Various types of abstractions have been used to model different aspects of a program. The most common abstraction is the control-flow graph, often called the program graph. The vertices represent statements or specific groupings of statements and branches represent transfer of control. Paige [13,14] has described the basic concepts as well as possible definitions of a vertex and the ways of partitioning program graphs according to the vertex definition used. Pratt [15-17] has extended the concept of the program graph to hierarchical relationships within programming languages and has also described a language for manipulating the hierarchical graph. Kunii, et al., [18-20] have built on the ideas of Pratt and others by describing recursive graphs (RG)--where branches as well as the vertices are decomposable. The control-flow graph has been used in the theory of global program optimization [21] and to aid in the restructuring of existing software [22-24]. A good discussion of control-flow analysis has been given by Hecht [25].

There have been other abstractions used to describe aspects of a program. Sholl and Booth [26] have described the use of computation structures to model the performance of a software system. A computation structure consists of two graphs, a control-flow graph and a data-flow graph. The data flow describes the possible movement of data objects within the software system. However, the graphical representation used is cumbersome. Kodres [27] has introduced data-flow graphs as a tool for the analysis of real-time systems. The graphs are used in test case generation, error analysis, analyzing parallel processing, and in program verification. Other abstractions used to model the data flow employ a program graph and other information to show where a data object is defined (set) and referenced; this information can be in the form of sets which are associated with vertices in the program graph. Fosdick and Osterweil [28,29] have used this approach to model the data flow within a program. Program optimization techniques [21] use a similar approach. Functional Programming [30], on the other hand, partitions a program into distinct

execution paths. This allows the conditions associated with a particular path to be more explicitly described.

Abstractions which describe data objects have also been used. However, most data object descriptions have been in terms of functions [31-34]. Earley [35] and Honig and Carlson [36] have attempted to formalize the description of the structure of aggregate data objects, independent of a particular implementation language. Jones [37] has used trees to model data types in conjunction with abstract data type, but not the structure of the data object.

There exist a number of different design methodologies, each using some form of program model upon which the design is built. Some methodologies, such as HOS [38] and SREM [39-41], use formally defined program models; others, such as SADT [42] and the Jackson Design Method [43], use more informal program models. SARA (System ARchitecture Apprentice) is a methodology which uses a Graph Model of Behavior (GMB) with the Dennis Data Flow (DDF) model to describe and analyze multilevel concurrent systems [44]. None of these methodologies are intended for the modelling of already existing software, however.

The program model used by HOS cannot be used to describe a program other than the one designed using the HOS methodology. To a limited extent, the program model used by SADT can be used to describe existing software, but the model is not formal and is therefore not suited to our purposes. PSL/PSA [45] uses a program model that is unable to describe relationships between levels of abstraction and it is unable to describe data structure. R-nets, used by the SREM methodology, comes closest to satisfying our requirements for the program model, yet it lacks a formal means of describing aggregate data structure in a well-defined way. Ramamoorthy and So [46] have given a good overview of the abstractions used for requirements and specifications of software systems.

4.1.2 Control Flow

The purpose of this abstraction is to describe all of the ways in which control can be transferred between statements within the program. Normally, the abstraction is a directed graph [13,14], where branches represent transfer of control and vertices represent sections of program code. In graphic form, it allows the programmer to get an overview of possible "flows" within the

program.

The type of control-flow graph is determined by the type of vertex. We identify three kinds of vertices for the control-flow graph: a module, a block, and a statement. The branches in a control-flow graph indicate possible transfers of control which can be made between nodes. Each branch has a set of conditions associated with it. The conditions are functions of the values of the program variables. For example, one condition could be "A > 2 and B = C", where A, B, and C are program variables.

We will illustrate the use of the control-flow graph at the module level in Figures 11-14. Figure 11 is a listing of subroutine LINER, where program blocks have been identified and numbered. Figure 12 is the control-flow graph for subroutine LINER, where vertices are blocks. Figures 13 and 14 show the comparable program listing and control-flow graph for subroutine PLOT. Note that in PLOT the blocks containing CALL LINER can be expanded into the control-flow graph of LINER.

In identifying blocks of a program, special attention needs to be given to the following situations: the IF statement, procedure calls, and DO loops.

A logical IF statement consists of two parts: a logical expression (evaluated as true or false), and an executable statement (which is only executed if the condition is true). Since the executable statement of the IF statement is not always executed, the whole IF statement cannot be a part of the same block. This can be seen in blocks 3 and 4 of subroutine LINER as shown in Figure 11.

Procedure calls are given separate blocks, as shown in block 9 of LINER (FORTRAN-callable routine ENCODE) in Figure 11 and in block 15 of PLOT (a call to subroutine LINER) in Figure 13. This is done in order to be able to decompose a routine into its constituent parts. For example, block 15 of subroutine PLOT in Figure 13 can be decomposed into the control-flow graph of subroutine LINER. For FORTRAN-callable routine ENCODE of block 9 of LINER shown in Figure 11, there is no need to decompose the block (if it were possible), but by assigning the routine call to its own block, it is easier to attach functional meaning to the block. This is true since the INPUT and OUTPUT sets defined for the block will be the same as the INPUT and OUTPUT sets for the routine.

```
                SUBROUTINE LINER(WEEK,VAR,SCORE,SYMBOL,LSYM,CNT,MONTH)
(1)             INTEGER WEEK,VAR,SYMBOL,HITE,ROW,COLUMN,AXISY,AXISX,YWIDTH,
               1        GRID(60,136),COLMSZ,ROWSZ,CNT,MONTH(3)
(2)             COMMON /BLOCK1/GRID,AXISX,AXISY,COLMSZ,ROWSZ,YWIDTH,MAXWKS
                XHITE=SCORE/0.2
(3)             HITE=INT(XHITE)
                DIF=XHITE-HITE
                IF(DIF.GE.0.5)
(4)             1               HITE=HITE+1
(5)             DIF=HITE-XHITE
                IF(DIF.GT.0.5)
(6)             1               HITE=HITE-1
(7)             COLUMN=(AXISY+2)+(VAR-1)*(MAXWKS+2)+(WEEK-1)
                IF(SYMBOL.EQ.1H* .AND. CNT.NE.0)
(8)             1               GO TO 10
(9)             ENCODE(1,200,NCNT) CNT
(10)            GRID(AXISX,COLUMN)=NCNT
                GO TO 12
(11)      10    GRID(AXISX,COLUMN)=1HC
(12)      12    GRID(50,COLUMN)=LSYM
(13)            DO 20 J = 1,3
(14)      20       GRID(AXISX-3-J,COLUMN)=MONTH(J)
(15)            IF(HITE.EQ.0)
(16)            1               GO TO 40
(17)            DO 30 ROW = 2,HITE+1
(18)               IROW = ROW-1
(19)      30       GRID(IROW+AXISX,COLUMN)=SYMBOL
(20)      40    CONTINUE
(21)      200   FORMAT(I1)
                RETURN
                END
```

Figure 11. Subroutine LINER partitioned into blocks.

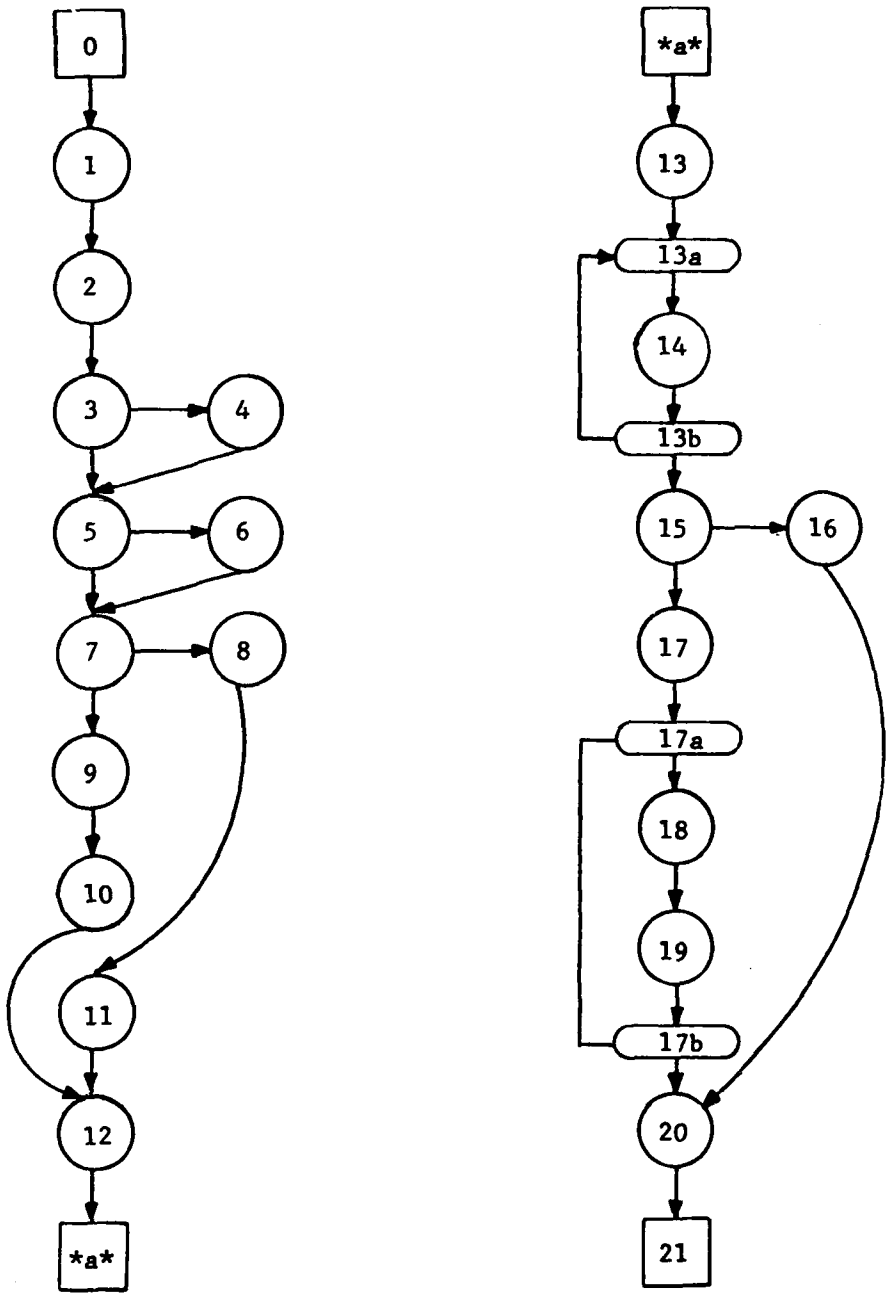


Figure 12. The control-flow graph for subroutine LINER

```
(1)             SUBROUTINE PLOT(MEANS,COUNTS,LABEL,N,NWEEKS,NLIN,NLOUT,DATE1)
                INTEGER COUNTS(21,7),VAR,SYMBOL,WK,LETTER(12),COUNT,CLMN,
               1        CLMN,MONTH(3),DATE1(21,12)
                REAL MEANS(21,7)
(2)             DATA LETTER/1HA,1HB,1HC,1HD,1HE,1HF,1HG,1HH,1HI,1HJ,1HK,1HL/
(3)             IF(LABEL.EQ.3HPOS)
(4)            1              NVAR=7
(5)             IF(LABEL.EQ.3HNEG)
(6)            1              NVAR=6
(7)             SYMBOL=1H*
                NCOUNT=NLIN
(8)             DO 20 VAR=1,N
(9)                NCOUNT=NCOUNT+1
                   NWKS=NWEEKS
(10)               DO 20 WK = 1,NWKS
(11)                  DO 10 J = 1,3
(12)       10            MONTH(J) = DATE1(WK,J)
(13)                  CLMN=WK
                      IF(MEANS(WK,VAR).EQ.9.0)
(14)           1              MEANS(WK,VAR)=0.0
(15)                  CALL LINER(CLMN,VAR,MEANS(WK,VAR),SYMBOL,
               1              LETTER(NCOUNT),COUNTS(WK,VAR),MONTH)
(16)       20         CONTINUE
```

Figure 13. Subroutine PLOT partitioned into blocks.

```
(17)              IDIF=NVAR-N-2
(18)              DO 50 IVAR=(N+1),(N+2)
(19)                 VAR=IVAR+IDIF
                     NCOUNT=NCOUNT+1
(20)                 DO 50 WK=L,NWKS
(21)                    DO 22 J=1,3
(22)          22           MONTH(J)=DATE1(WK,J)
(23)                    CLMN=WK
                        IF(MEANS(WK,VAR).EQ.-1.0)
(24)              1              GO TO 26
(25)                    IF(VAR.NE.NVAR)
(26)              1              GO TO 30
(27)                    SYMBOL=1H*
                        LSYM=1H*
                        GO TO 40
(28)          26        MEANS(WK,VAR)=0.0
(29)                    SYMBOL=1HX
                        LSYM=1HX
                        GO TO 40
(30)          30        LSYM=LETTER(NCOUNT)
(31)          40        CALL LINER(CLMN,VAR,MEANS(WK,VAR),SYMBOL,LSYM,
                  1                COUNTS(WK,VAR),MONTH)
(32)          50        CONTINUE
(33)              NLOUT=NCOUNT-1
(34)              RETURN
                  END
```

Figure 13. Subroutine PLOT partitioned into blocks (cont.).

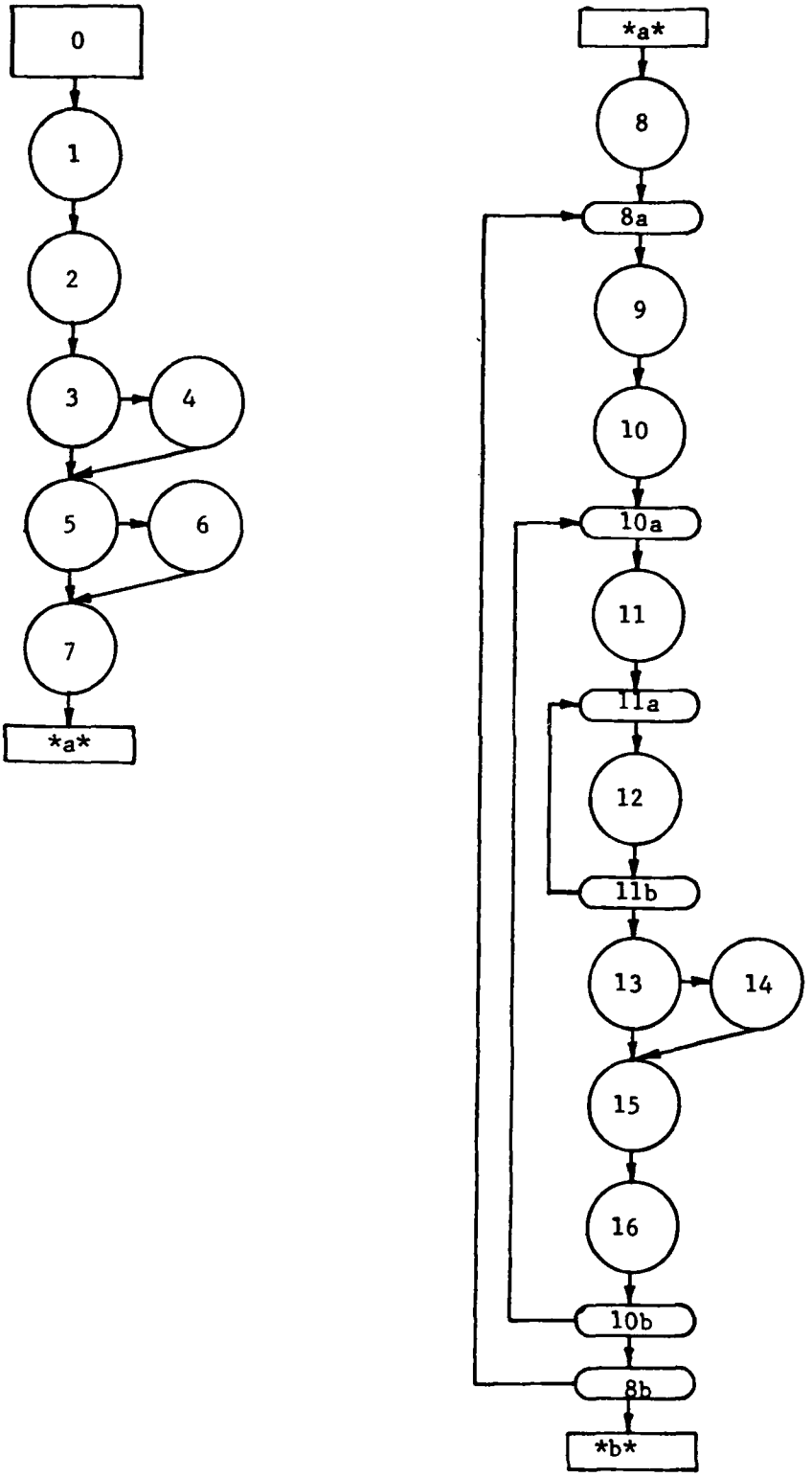


Figure 14. The control-flow graph for subroutine PLOT.

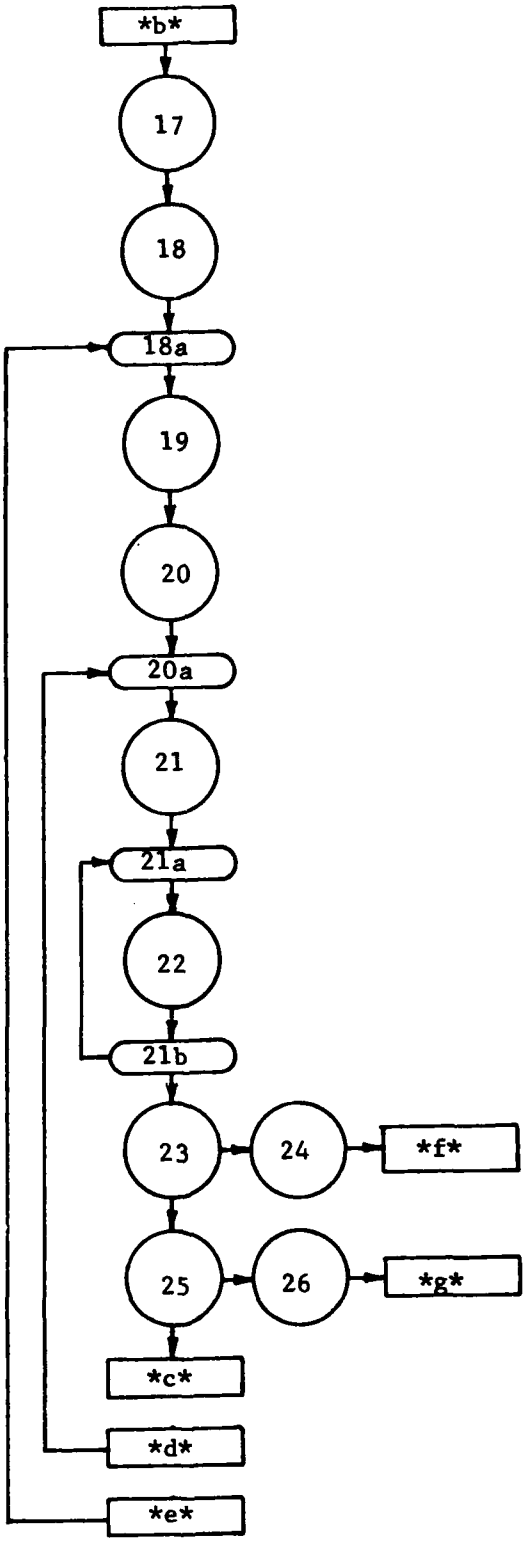


Figure 14. The control-flow graph for subroutine PLOT (cont.).

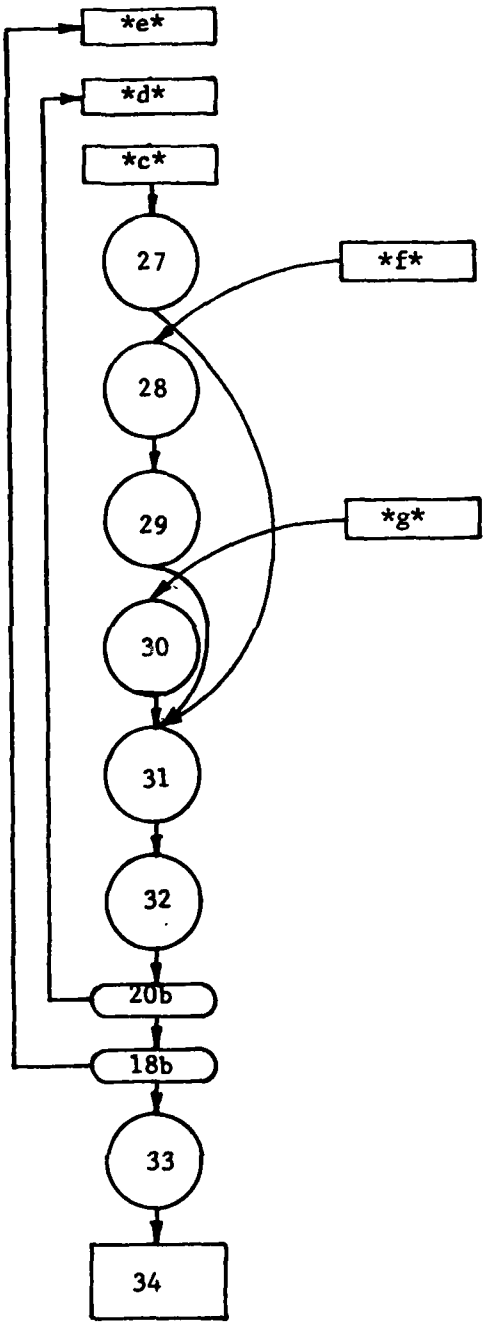


Figure 14. The control-flow graph for subroutine PLOT (cont.).

For the DO loop construct, we add two special nodes, such as nodes 13a and 13b in the control-flow graph for LINER shown in Figure 12, and we replace the DO statement itself. The following is an example from blocks 13 and 14 of LINER shown in Figure 11:

```
(13)      DO 20 J = 1,3
(14)  20      GRID(AXISX-3-J,COLUMN) = MONTH(J)
```

is replaced by

```
(13)      J = 0
(13a) 19      J = J + 1
(14)          GRID(AXISX-3-J,COLUMN) = MONTH(J)
(13b) 20      IF(J.LT.3) GO TO 19
```

The reason for doing these alterations is to more explicitly describe the looping structure which the DO loop construct represents. The actual statement replacement need not be visible to the maintenance programmer, but would be used in the analysis of the program's structure and error characteristics.

There are some other aspects of the control-flow graphs which need explanation, such as the role of the subroutine-statement vertex and the COMMON-statement vertex. Both of these vertices can be considered as input-output "ports". Although no variables in these two vertices are part of an executable statement, we still construct the input set and the output set for both vertices. This allows us to better define the data flow within the module (see the data-flow graph for subroutine LINER).

Since modules can be decomposed into blocks and blocks decomposed into statements, there is an imposed hierarchy within the control-flow graph. The uppermost level of this hierarchy is the main program module. The lowest level of the hierarchy contains individual statements.

4.1.3 Data Flow

The purpose of this abstraction is to describe all the possible ways in which data can be transferred between program statements. Not as common as the control-flow graph, a data-flow graph is a directed graph, where branches represent transfer of data and vertices represent sections of code. It allows the programmer to see how individual data objects are used in the program, and where they are defined (set) and referenced.

The data-flow graph we have defined uses the three vertex types described for the control-flow graph as well as a fourth: the substatement. A substatement vertex contains one of the syntactic elements of the statement (e.g. a variable, an operator). While the statement vertex treats a statement as a block box, a graph at the statement level (vertices are substatements) reveals the syntactic structure of the statement. An example of a data flow graph at the statement level, where substatements are vertices, is given in Figure 15.

A = 1 + B

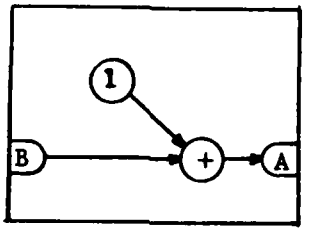


Figure 15. A data-flow graph at the statement level.

The graphical notation is similar to that used in [19] to describe systems.

For the data-flow graph we have defined the following associated sets:

P_i = the set of input parameters
P_o = the set of output parameters
G_i = the set of global input variables
G_o = the set of global output variables
S = the set of secondary (local) variables

The associated sets for subroutine LINER are:

P_i = {WEEK,VAR,SCORE,SYMBOL,LSYM,CNT,MONTH}
P_o = { }
G_i = {AXISX,AXISY,MAXWKS}
G_o = {GRID}
S = {XHITE,HITE,DIF,COLUMN,NCNT,J,ROW,IROW}

The sets are constructed for modules. For blocks and statements, only simple input and output sets are constructed.

The branches of a data-flow graph are labelled to represent the name given to the set of data objects which "flows" along the branch. Since such a set may contain more than one variable, the set itself can be thought of as a type of data object which can be decomposed into its constituent elements. Since data objects can be placed in these aggregate sets, it reduces the number of "data objects" which are part of the data-flow graph. If more detail is desired, the desired labels can be decomposed to reveal individual data objects.

In a data flow graph, each vertex can be considered as a "processing unit" and each branch a "pipeline" through which data objects move. Since three of the four vertex types used are the same as those used in the control flow graph, there is a similar facility for decomposition. However, in the data-flow graph, there is the added possibility of decomposing the branches in terms of the branch sets.

The input and output sets for the vertices in a data-flow graph can be easily constructed. For example, the data-flow graph and the branch sets for a section of subroutine LINER shown in Figure 11 are given in Figures 16 and 17. The branches going into block 10 in the data-flow graph of subroutine LINER shown in Figure 16 represent the sets which contain the input elements to block 10, and the branches leaving block 10 represent the output elements. Hence, branch sets 11, 12, and 10, which include AXISX, COLUMN, NCNT, are subsets of the input set to block 10, and branch set 17 constitutes the output set of block 10.

4.1.4 Data Object Structure

The purpose of this abstraction is to describe the logical structure of data objects. This is necessary since the structure of data objects may change. In order to describe such a change, a well-defined abstraction is necessary. Since a data object structure can have a great deal to do with the types of processes defined within a program, such as the procedures defined for a table structure, a clear definition of the structure of data objects can help the programmer understand the program.

We define a type classification scheme for data object structure and an associated graphical representation using the following six questions, independent of the implementation language used. Of the following six

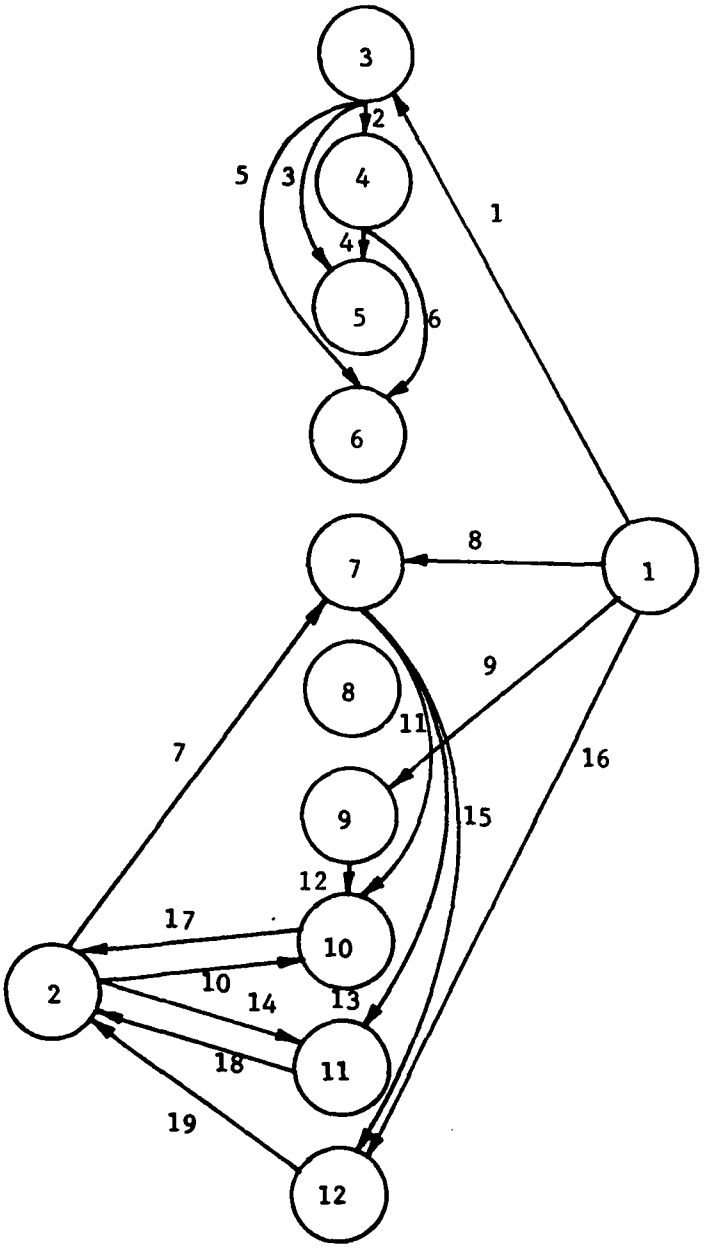


Figure 16. The data-flow graph for blocks 1-12 of LINER.

Branch Number	Branch Set
1	{SCORE}
2	{HITE}
3	{HITE,XHITE}
4	{HITE}
5	{HITE}
6	{HITE}
7	{AXISY,MAXWKS}
8	{VAR,WEEK,SYMBOL,CNT}
9	{CNT}
10	{AXISX}
11	{COLUMN}
12	{NCNT}
13	{COLUMN}
14	{AXISX}
15	{COLUMN}
16	{LSYM}
17	{GRID}
18	{GRID}
19	{GRID}

Figure 17. The branch sets for the data-flow graph in Figure 16.

questions, the last five questions were suggested by Honig and Carlson [36].

1) Aggregate: (yes, no)
2) Homogeneous: Do the members have the same structure? (yes, no)
3) Basic: Are the members atomic? (yes, no)
4) Ordered: Are the members ordered? (yes, no)
5) Number: What is the nature of the aggregate cardinality? (fixed, variable but bounded, unbounded)
6) Identification: How are the members identified? (number, name, pointer, none)

These questions allow us to classify a data object independent of any particular programming language. In order to further identify a data object's structure we add the following secondary set of questions:

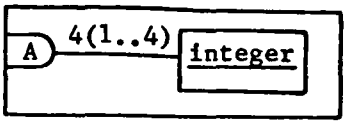
2) Homogeneous
   2.1) If not homogeneous, how many structures are represented?
3) Basic
   3.1) What is (are) the data structure(s)?
4) Ordered
   4.1) If ordered, what is the sorting variable?
5) Number
   5.1) If fixed or bounded, what is the bound?
6) Identification
   6.1) If identified by number, name, or pointer, what is the identifier type for a given element?

Since a data structure can be described in terms of other data structures, there is the possibility of describing hierarchical or recursive data structures.

Figure 18 shows some examples of data object structures represented as type graphs and their associated classifications. The underlined words are considered to be atomic (in the case of integer and string) or reserved words (in the case of fixed and number). Note that in Figure 18(b), there is a node C which is not atomic.

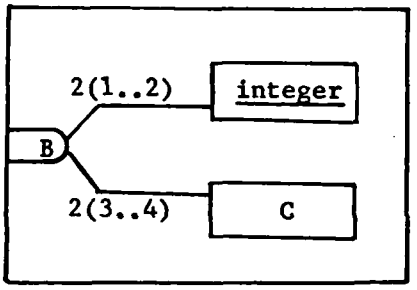
The types of vertices for the type graph are different from vertices in the data-flow or control-flow graphs. In the three sample graphs, there are two types of vertices: Name and element. For example, in

(a) A FORTRAN array A with four integer elements



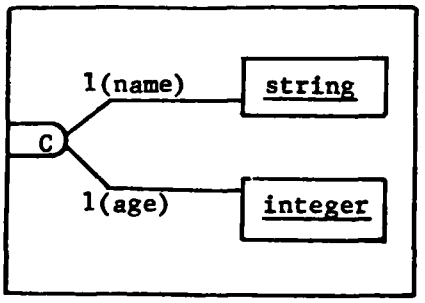
Aggregate: yes
Homogeneous: yes
Basic: yes,integer
Ordered: yes,index
Number: fixed,4
Identification: number,
(1..4)integer

(b) A non-homogeneous array B



Aggregate: yes
Homogeneous: no,2
Basic: no,integer,C
Ordered: yes,index
Number: fixed,4
Identification: number,
(1..2)integer
(3..4) C

(c) A record structure C with two fields: name and age



Aggregate: yes
Homogeneous: no,2
Basic: yes,string,integer
Ordered: no
Number: fixed,2
Identification: name,
(name)string
(age)integer

Figure 18. Some graphs of data object structures and their associated classifications.

Figure 18(c) vertex C is a name vertex and the two rectangles (string and integer) are element vertices. The branches in a type graph do not have associated directions and show logical associations between vertices and allow the associations to be described through the use of labels.

4.1.5 Execution Flow

The purpose of this abstraction is to identify all possible program paths in the program and their associated conditions (if possible). Actually, the execution flow is another way of describing the information contained in the control-flow graph. Although it is not as important as the control-flow graph, the execution flow is useful when analyzing the program's structure. The most common representation of execution flow is simply a sequence of vertices with some notation to indicate which vertices can be executed more than once. For example abc*d represents the execution sequence with a, b, c, and d as vertices, where c can be repeated any number of times.

The vertices used in an execution-flow graph are the same as those used in the control-flow graph. This allows the vertices to be decomposed in the same way as done for a control-flow graph. The branches in an execution-flow graph merely indicate the order in which the sequence occurs since there are no decision points in an execution-flow graph. Figure 19 shows a simple execution-flow graph.

Figure 19. A simple execution-flow graph representing the sequence abc*d.

4.2 Use of the Abstractions

In order to describe a modification to an existing program, we construct

the four abstractions for the program. This constitutes the program model. Any change to the program is then described in terms of the model. Therefore, it is necessary to define a set of primitive operations which can be used to modify the program's abstractions. We have identified the following primitive operations:

1) add a vertex,
2) delete a vertex,
3) add a branch,
4) delete a branch,
5) add a branch label,
6) delete a branch label,
7) add a variable to a branch set,
8) delete a variable from a branch set,
9) add a branch set to a branch,
10) delete a branch set from a branch.

For each abstraction, a particular set of operations apply.

For the control-flow graph, only operations 1) through 4) apply. For the data-flow graph, all ten operations apply. The data-object-type graph would use the first six operations, and the execution-flow graph (like the control-flow graph) would use the first four.

The first six operations can be formalized as follows:

1) INV(R,X,y): Insert vertex x representing system X in the graph R as a subvertex of y
2) DLV(R,x): delete vertex x and all of its subvertices and all branches connected to these vertices from the graph R
3) INB(R,A,[x,y]): insert branch a representing association A and directed from x to y in the graph R
4) DLB(R,a): delete branch a and all of its subbranches from the graph R
5) IBL(R,a,ℓ): label branch a in graph R with label ℓ
6) DBL(R,a): delete label from branch a in graph R

In a similar manner, operations can be formalized for the other four primitive operations. The above operations are similar to those proposed by Kunii and Harada [19].

4.2.1 An Outline of a Methodology for the Specification of Program Modification Proposals

The following is an outline of a methodology which can be used to specify a possible modification:

1) Construct the program model (the four abstractions).
2) Determine the type of change to be made in terms of one of the abstractions (control-flow, data-flow, data-object-structure).
3) Locate the lowest level of abstraction at which the entire change is "visible".
4) Describe the proposed change in terms of the primitive operations defined on the graph.
5) Determine the effects of the change on all of the abstractions in the program model.

In most cases the change will be classified as a control-flow or data-flow change. However, when the main alteration is for a data object structure, the change should be classified accordingly. After the change is classified, the programmer should go to the corresponding graph to determine the level at which the change is to take place. This level must be the lowest level of abstraction at which the entire change can be described within one vertex. This vertex is called the unit of change. For example, if we desire to alter a data structure, we would find the lowest level in the associated graph in which the entire change is to be described. If the data structure is a table of records (a record having three fields, each of a different type) and we wish to change two of the three fields in the record type, then the record type graph would be the unit of change.

It is possible that once the programmer has identified the unit of change, the alterations could be made by a text editor. The text editor would have the following characteristics:

1) The unit of change would reside in the workspace of the text editor.
2) Any modification to the workspace would be translated into the set of primitive operations for the given graph automatically by the system. The system would keep track of modifications so that any edition of the program could be reconstructed.
3) The text editor would not actually be able to change the program model.

5) The purpose of the text editor would be to translate "ordinary text modifications" into a set of formal primitive operations which would constitute the specification of the modification proposal.

6) The set of formal primitive operations (the specification of the proposed modification) would then be used to actually modify the program in terms of the program model.

4.2.2 An Example

Suppose that the user wishes to change the plotting symbol used in the first 16 blocks of subroutine PLOT shown in Figure 13 by changing the '*' to '0'. We use the proposed methodology to specify the change to be made in terms of primitive operations defined on the program model.

Assuming that the program model has been constructed, we must first determine the type of change to be made. Since the change will be made to a substatement within the first statement of block 7, the type of change will be a data-flow change. This is our choice since only the data-flow abstraction is able to describe the relationships which exist between substatements.

The lowest level of abstraction at which the entire change is "visible" is at the statement level, where substatements are the vertices of the statement level, data-flow graph. Therefore, the first statement in block 7 of subroutine PLOT shown in Figure 13 is the unit of change for the modification. Figure 20 shows the graph for this unit of change before and after the modification.

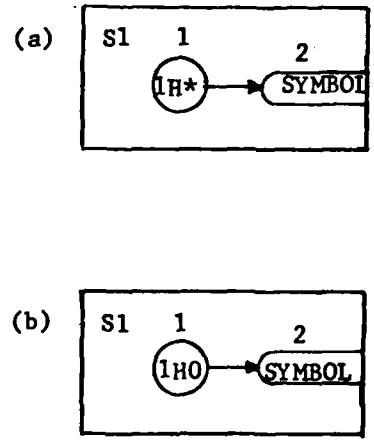


Figure 20. An example illustrating the use of a data-flow graph at the statement level to describe a statement before (a) and after (b) a change.

This change would be expressed as the following sequence of commands operating on graph S1 of block 7 in subroutine PLOT:

DLV(S1,1)	delete vertex 1 in graph S1
INV(S1,X,1)	insert vertex 1 representing system X into graph S1
INB(S1,A,[1,2])	insert a branch from node 1 to node 2 representing association A into graph S1

The operation DLV(S1,1) deletes vertex 1 from graph S1 and all of the branches which were attached to vertex 1. INV(S1,X,1) places the vertex representing system X inside the box representing graph S1. After this operation, there is a vertex representing X with no branches attached. In order to add the necessary branch from vertex 1 to vertex 2, INB(S1,A,[1,2]) must be executed. This adds the branch representing association A from vertex 1 to vertex 2. System X would be the character string '1HO', and association A is essentially null since an association label in this case is not necessary. The effects of this change on the other three abstractions are nil since no data structure or control flow are affected. The change is felt only within statement 1 in block 7 of subroutine PLOT.

As noted before, the programmer should be able to make the textual change to a line of code (in this case, to a substatement) and have the text editor construct the corresponding sequence of primitive operations on the program model. It is this resultant sequence which would constitute the formal specification of the proposed modification. All of this implies that the text editor would have access to the structural information contained in the program model. In other words, the text editor would have to "know" the level at which the change is to occur and what objects (module, block, statement, or substatement) are being inserted or deleted.

4.3 Conclusion

When an activity is to be formally described, all of the objects and relationships involved in the activity must necessarily be well-defined. This is the case for the "activity" of describing a given program modification in formal terms--all of the objects and relations involved in the modification must be well-defined.

Most of our work in this area has involved the identification of the pertinent objects and relations affected by a program modification. We have tried to identify those particular objects and relations which are common to all programs no matter what implementation language is used. Among those relations and objects, we must decide which could be formalized and which would be most useful. Consequently, we have chosen the module, the block, the statement, and the substatement as objects and the abstractions of control flow, data flow, execution flow, and data object structure as representations of the relationships among these objects.

More work needs to be done in order to formally describe the relationships among the four identified abstractions. Furthermore, more work needs to be done in order to identify the primitive operations for all four abstractions. So far we have only defined a few operations for the control-flow and data-flow abstractions.

With the tremendous amount of information contained in a set of program abstractions, we need a database system to manage the abstractions. This is especially true when using the abstractions to describe large-scale programs. A relational database [47] seems to be the best type of database to use with our methodology. We have considered a hierarchical database model before, but found it unable to effectively describe all aspects of the four abstractions. Furthermore, unlike the relational model, the hierarchical model has no formalized method of describing operations.

Finally, we have identified some of the essential aspects of a methodology for the formal description of a given program modification. Although it needs to be formalized, it appears feasible and language-independent. If incorporated with a database system and a text editor, it should provide a workable and straightforward technique for the formal description of given program modifications.

5.0 EFFECTIVE TESTING FOR SOFTWARE MAINTENANCE

As mentioned before, the fourth phase of the maintenance process is to test the modified program. However, very little results are available in this area. Although there have been some considerations on this type of testing [48-50], the results neither are practical, nor cover the generation of test cases for modified programs. In fact, testing the modification performed on

a large-scale program during the maintenance phase has normally been regarded as an extention of testing performed during the development phase. However, because the testing environments for the development phase and the maintenance phase are different, we need a different approach to testing in the maintenance phase. Therefore, we have considered to establish a testing strategy for validating the modification performed on a large-scale program and ensuring that the modified program is reliable. In this section, we will present our results in this area so far obtained.

5.1 Errors Likely Encountered in the Maintenance Phase

Our testing strategy in the maintenance phase is intended to detect two major kinds of errors. The first kind is the residual errors in the program which become critical errors due to the modification of the program. The second kind is the errors introduced as a result of the modification. The ripple effect analyzer presented in Section 3 can only detect a subset of the errors of the second kind, i.e. the ripple effect of the modification. Before we present our testing strategy, let us first discuss those errors which can be encountered in the maintenance phase.

5.1.1 Residual Errors Activated by Program Modifications

It is well known that large-scale software systems always contain a number of residual errors simply because there exists no software design methodology that can ensure that the large-scale software systems generated by the design methodology are error-free. These residual errors are called dormant errors because they do not affect the normal operation of the software systems; otherwise, they would have likely been detected. However, due to certain program modifications, some dormant errors may become activated. For example, if some program modifications cause the change of execution paths so that the execution paths in the modified program may involve certain dormant errors, then these dormant errors will become active errors. Changes of input domains of certain execution paths [51] and loss of compensation effect of certain errors due to program modifications are other examples of this type of errors.

5.1.2 Errors Introduced by Program Modifications

We cannot assume that the maintenance programmers always do a perfect job for program modifications, and hence new errors may be introduced during the

modifications. This kind of errors can be considered in the following categories.

5.1.2.1 Errors Introduced by Making Modifications

In the software development phase, the programmers may introduce errors in any development stage. During the maintenance phase, the programmers may also introduce errors during the generation of modification specification and the implementation of the modifications. In fact, the chances of introducing errors by making modifications in the maintenance phase are often bigger than that in the development phase because of incomplete documentation, misunderstanding by the maintenance programmers and lack of effective software maintenance methodology. These errors could be any of the three types, each associated with one of the three abstractions: data flow, control flow and data object structure, as discussed in Section 4.1.

5.1.2.2 Inconsistencies Introduced by Program Modifications

Because of program modifications, there is ripple effect from the locations of modifications, and certain data flow and control flow may be changed. These changes may cause inconsistencies in the program if all their ripple effect is not taken care of. In the following subsections, we will discuss various types of errors belonging to this category.

5.1.2.2.1 Inconsistencies Due to Data Flow Changes

Suppose a modification involves several variables, which may be used to define another variable, say A. In turn, variable A may be used to define other variables. In this way, the effect of the modification is carried through data flow. The ripple effect analyzer discussed in Section 3 can trace the ripple effect due to data-flow changes and identify the program blocks containing the variables which may need re-examination. Data-flow inconsistencies can be defined with respect to the data flow abstraction discussed in Section 4.1.

5.1.2.2.2 Inconsistencies Due to Control Flow Changes

These inconsistencies are caused by the changes in conditional statements (IFs), transfer statements (GOTOs), etc. It is usually more difficult to identify the inconsistencies due to control flow changes than those due to the data flow changes. For example, consider a segment of the program

shown in Figure 21(a). When I is 0, the statement K = M/I is in error, and the maintenance programmer may correct the error as shown in Figure 21(b) by changing the statement from "IF(I.LE.0) GO TO 10" to "IF(I.LT.0) GO TO 10". This

```
      :
      :
    READ, I
    IF(I.LE.0) GO TO 10
      :
      :
    K = L/I
      :
      :
    GO TO 20
10  K = M/I
      :
      :
```

(a)

```
      :
      :
    READ, I
    IF(I.LT.0) GO TO 10
      :
      :
    K = L/I
      :
      :
    GO TO 20
10  K = M/I
      :
      :
```

(b)

Figure 21. An example illustrating an inconsistency caused by a control flow change: (a) the program with an error before modification, and (b) the program after a possible correction for the error.

modification avoids the error encountered in the statement K = M/I, but may cause another error in the statement K = L/I because there may be a control flow from the modified statement to the statement K = L/I. When I is 0, the statement K = L/I is erroneous. This type of errors cannot be covered by the present ripple effect analyzer discussed in Section 3, although it can be classified in terms of the control-flow abstraction discussed in Section 4.1.

5.1.2.2.3 The Constant Problem

It is a good programming practice to limit the use of constants in developing programs, and a data item in a program can be defined in terms of an expression. Let us consider the case as shown in Figure 22. A data item

```
      .
      .
      .
INDEX = 2
      .
      .
      .
ADDR = 5 + INDEX
      .
      .
      .
```

Figure 22. An example to show the constant problem.

INDEX is defined in terms of an expression consisting of only a constant and is used to define another data item ADDR. Since it is not an error to use the value which is assigned to INDEX to define ADDR, "ADDR = 5 + INDEX" may be written as "ADDR = 7". When "INDEX = 2" is modified to "INDEX = 3", the definition of ADDR must be adjusted to reflect this change. However, there is no effective way to adjust it because INDEX is no longer used to define ADDR. The failure of adjusting such a definition introduces an error in the program. This type of errors can be defined in terms of the data-flow abstraction at the statement level, as described in Section 4.1.

5.1.2.3 Errors Introduced Due to Imperfect Correction of Identified Errors

After a modification, a subset of the inconsistencies and the activated errors are identified. Since we cannot assume that the maintenance programmer is able to correct all the errors, it is reasonable to assume that after all the corrections are completed, the program may contain the following types of errors: newly introduced errors, unidentified inconsistencies, and unidentified activated errors. An error of this category could be any one of the three types, each associated with one of the three abstractions: data flow, control flow, and data object structure as discussed in Section 4.1.

5.2 A Testing Strategy in the Maintenance Phase

In this section, we will present a testing strategy for validating programs after their modifications. We will give an overview of the testing strategy and discuss which types of errors may be detected by the testing strategy. A part of the testing strategy covering module testing which we have developed will be presented in the next section.

When testing is performed, at least the following three factors must be considered: 1) which basic elements of the program are to be tested, 2) how these basic elements are to be tested, and 3) how thorough a program is tested, where basic elements we consider in testing modified programs are intramodule control flows, intermodule control flow and program functions.

For the first factor, we will use three major testing schemes, each testing some aspect of the program: module testing tests intramodule control flows, integration testing tests intermodule control flows, and system function testing tests the program against its functional specifications.

For the second factor, we will use two kinds of techniques to derive test cases. The first technique, called the structural testing, tests the internal structure of a program, especially its control flow. The second technique, called the functional testing, tests the functional specification of the program.

The third factor depends on the testing criterion used. Many testing criteria have been proposed, but most of the criteria are for structural testing, not for functional testing.

As mentioned before, we need to test the functions and control flow of a program after its modifications. It is noted that although data flow normally cannot be tested, it can be analyzed by the ripple effect analyzer discussed in Section 3. Program functions and control flow cannot be tested by anyone of the testing schemes mentioned before, and hence it is necessary to consider the use of a combination of the three major testing schemes for testing the functions and control flow of a modified program. Before considering this, we would like to discuss which technique is suitable for which scheme. For system function testing of large-scale programs, it is not feasible to use structural testing to derive test cases because a large-scale program is huge in size and complex in structure. Instead, test cases are derived by

analyzing the functional specifications of the program, and such specification information is assumed to be available in the maintenance phase. In other words, we use the functional testing technique for system function testing. For module testing and integration testing, we cannot rely upon the information on the specifications for modules or module interactions, since they are often not available in the maintenance phase. Therefore, we need to derive test cases for module testing and integration testing from the structure of a program, i.e. we use the structural testing technique for module testing and integration testing.

A testing criterion commonly used for testing a module is that each DD-path in the module is traversed at least once [52,53], where a DD-path of a module is a series of branches in the module such that

1) the first branch starts from either the entry point of a module or a decision point,
2) the last branch terminates either at a decision point or the exit point of a module, and
3) there are no decision points on a DD-path except at its both ends.

The test coverage of a testing strategy for a program depends how many modules in the program and how many DD-paths in each modules are tested.

Our strategy for testing modified programs is to use system function testing, followed by module testing and then integration testing. The rationale for having system function testing as the first portion of our testing strategy is that the test coverage of system function testing is the largest among those of the three testing schemes. However, because test cases for system function testing are not generated from the program structure, but from the program functional specification, we only know which DD-paths of which modules of the program are traversed after the system function testing is performed. Although the test coverage of system function testing is the largest among those of the three testing schemes, it is usually not sufficient to cover all the intramodule and intermodule control flows. Hence, we need to use module testing and integration testing after applying the system function testing. We use the module testing before the integration testing because of the following reasons: module testing is less complex than integration testing, and the information derived from the module testing, especially

regarding the intramodule control flows, is very useful to simplify the task of performing the integration testing.

It is clear from the above discussion that our testing strategy for modified programs is to test the control flows and program functions affected by the modifications, and that we use the ripple effect analyzer discussed in Section 3 to cover the changes of data flows of the modified programs affected by the modifications.

5.3 Module Testing

Module testing in the maintenance phase is to test intramodule control flows of modified modules affected by modifications. In other words, we need to test all the modified DD-paths, newly added DD-paths, and the DD-paths reachable to or from them. The testing criterion for our module testing scheme is to traverse each of these DD-paths at least once.

Our module testing scheme based on this testing criterion is heuristic and can be described as follows: Let $\mathcal{D}$ be the set of all modified DD-paths, newly added DD-paths and the DD-paths reachable to or from them in the module. Let a level-i path i = 0,1,2,..., in a module be defined as follows:

1) A level-0 path in a module leads from the entry point to the exit point of the module without using any DD-path more than once.
2) A level-i path, i = 1,2,..., leads from an alternative predicate outcome along some level-(i-1) path through a set of DD-paths not present on any lower level path, and terminates on the level-(i-1) path at a point earlier than the original DD-path. A level-i path, i = 1,2,..., represents iteration "over" a level-(i-1) path [52].

The derivation of all the level-i paths, $i \geq 0$, in a module can be represented by a tree T. There exists a software tool RXVP [52] that can generate all the level-i paths, i = 0,1,2,..., automatically. The subtree of the tree T containing all the level-i paths, i = 0,1,2,..., with a level-0 path P_j as its root, is called the subtree of P_j and is denoted by T_j. With these definitions, our module testing scheme can be presented as follows:

1) Select a subtree T_j containing the largest number of elements in $\mathcal{D}$. If there are two or more such subtrees, arbitrarily choose one of them.
2) Select a path from the entry point to the exit point consisting of only the branches contained in any of the level-i paths in the subtree T_j such

that the path contains the largest number of elements in $\mathcal{B}$ as a test path. Use any of the available techniques [54-56] to generate a test case to traverse this test path. If it succeeds, delete all the elements in $\mathcal{B}$, and let the remaining set be $\mathcal{B}'$. If the generation of a test case to traverse this test path fails, select another path with the next largest number of elements in $\mathcal{B}$ as the test path. If the number of elements in $\mathcal{B}$ covered by this test path is greatly reduced, it may be worthwhile to change the subtree T_j to a subtree T_k which has the same or the next largest number of elements in $\mathcal{B}$, and repeat this process until some elements in $\mathcal{B}$ are covered by a test case, and reduce the set $\mathcal{B}$ to $\mathcal{B}'$.

3) Repeat Step 2) until $\mathcal{B}$ becomes empty or it is decided that all the remaining elements in $\mathcal{B}$ are nontraversable. In the latter case, the modified module needs to be examined so that the existence of the nontraversable DD-paths can be explained [57].

Let us demonstrate this module testing scheme to one of the modules in a JOVIAL program called DEMO, a revised version of a demonstration program BLISTIC. This module is called FIND and searches a data base to see whether or not it contains a given set of data. Its source code, directed graph of DD-paths, the tree T, and a table listing the DD-paths which make up each level-i path are given in Figure 23 and Table 1. We use the notation (T_j, N) to keep track of N, where N is the number of elements in the set $\mathcal{B}$ which is contained in the subtree T_j.

Suppose that DD-path$_1$ and DD-path$_8$ are modified, and we have $\{(T_1,4), (T_2,5),(T_3,6),(T_4,7),(T_5,5)\}$. We select T_4 to construct a test path. The derived test path is 1-3-5-7-9-10-3-5-7-9-10-3-5-7-9-11, and is executable. As a result of this derivation of the test path, we now have $\{(T_1,0),(T_2,0), (T_3,0),(T_4,0),(T_5,1)\}$. Next we select T_5 and the derived test path is 1-3-5-7-8, which is also executable. Now since we have exhausted all the elements in $\mathcal{B}$, we terminate the scheme.

6.0 QUALITY FACTORS FOR SOFTWARE MAINTAINABILITY

As mentioned in Section 2.0, there are several software quality factors affecting software maintainability. We have developed a stability measure and a measure for understandability of programs. We will briefly discuss these results.

D-nodes

```
A        PROC FIND(F1,W1,W2,T1 = FLAG)
         ITEM F1 F $
         ITEM W1 F $
         ITEM W2 F $
         ITEM T1 F $
         ITEM FLAG I 36 S $


         BEGIN
         FLAG = 1 $
         FOR I = 0, 1, 2 $
         BEGIN
         IF F1 NQ THRST($I$) $
B            TEST $
         IF W1 NQ LOADWT($I$) $
C            TEST $
         IF W2 NQ FLWT($I$) $
D            TEST $
         IF T1 EQ BURNTM($I$) $
E            GO TO HIT $
F        END
         RETURN $
HIT.        FLAG = 0 $
         RETURN $
G        END
```

Figure 23. An example to illustrate the module testing scheme. (a) The source code of module FIND.

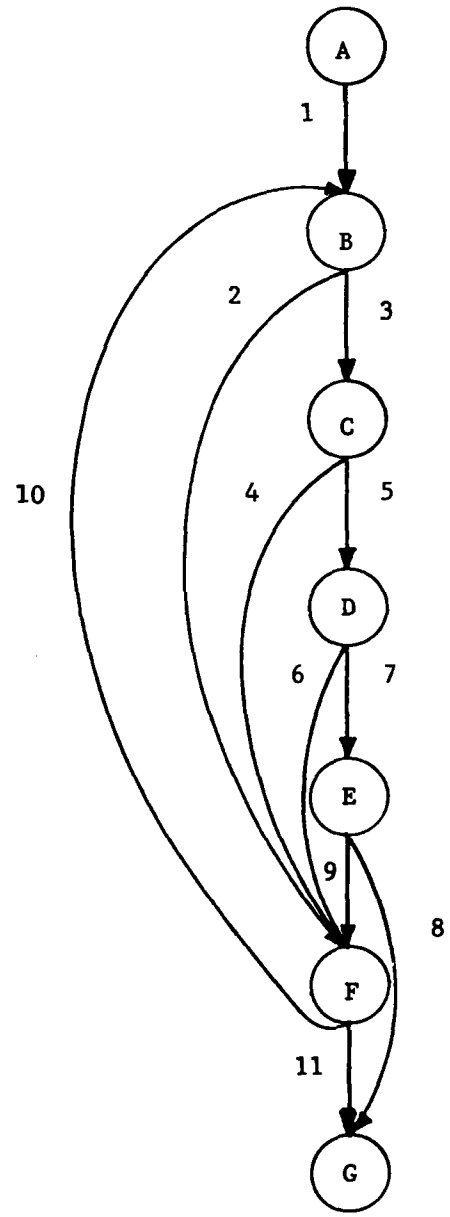
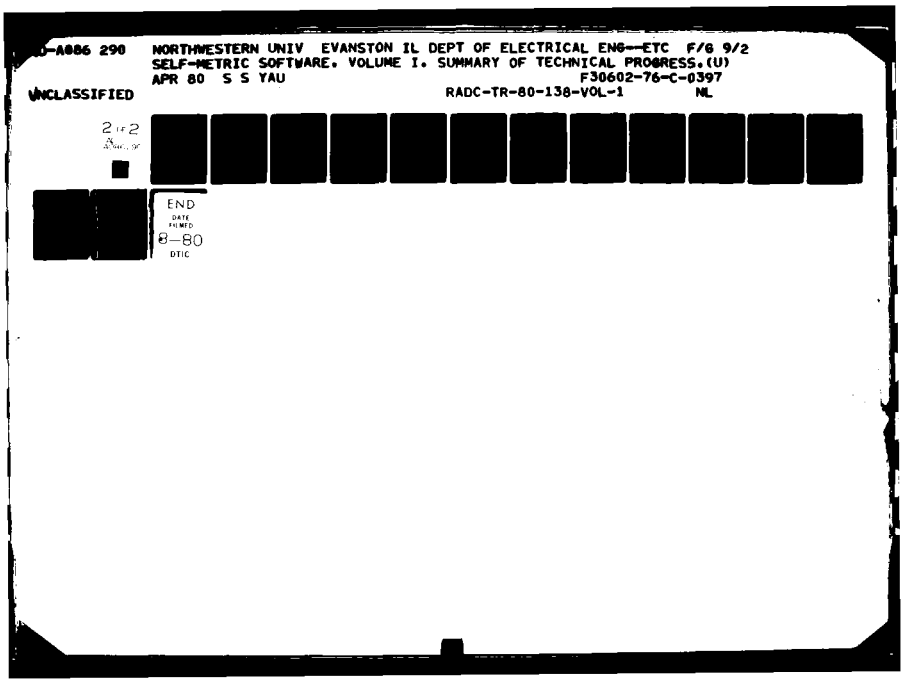
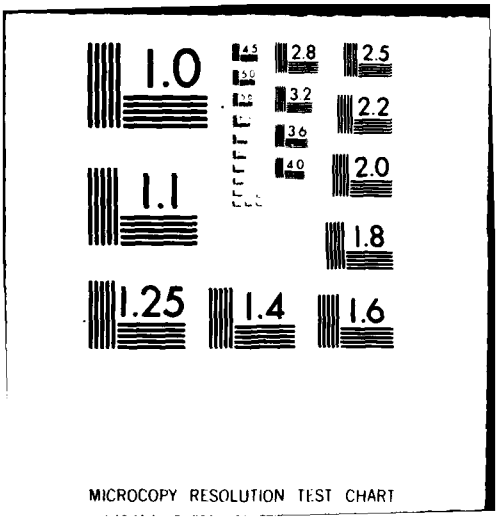


Figure 23. (b) The directed graph of DD-paths for module FIND.


D-A086 290
NORTHWESTERN UNIV EVANSTON IL DEPT OF ELECTRICAL ENG--ETC F/G 9/2
SELF-METRIC SOFTWARE. VOLUME I. SUMMARY OF TECHNICAL PROGRESS.(U)
APR 80 S S YAU
F30602-76-C-0397
UNCLASSIFIED
RADC-TR-80-138-VOL-1
NL
2 OF 2
END
DATE FILMED
8-80
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART

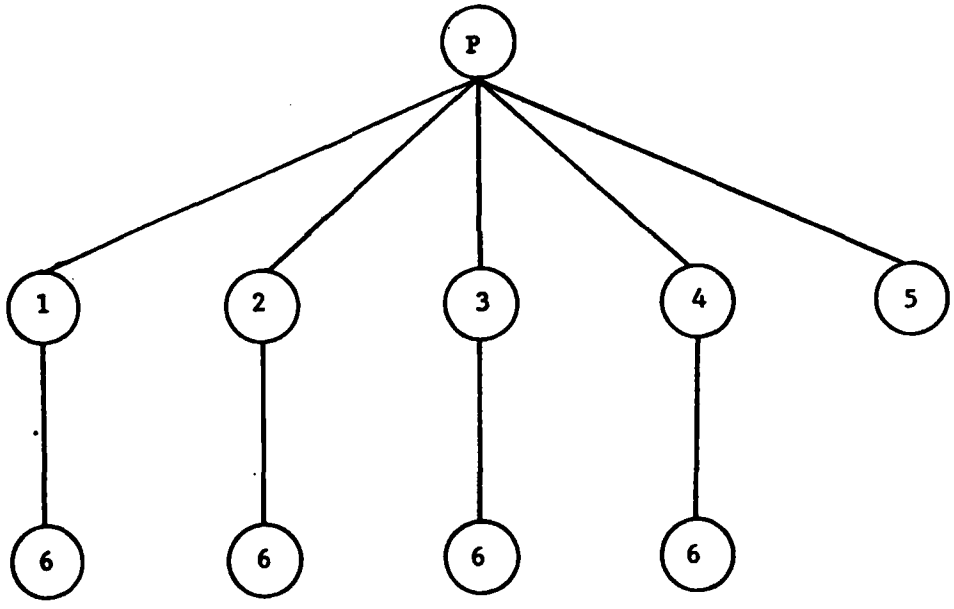


Figure 23. (c) The tree T of all level-i paths for module FIND.

Level-i path no.	i =	DD-paths belonging to it	Initial node	Terminal node
1	0	1, 2, 11	A	G
2	0	1, 3, 4, 11	A	G
3	0	1, 3, 5, 6, 11	A	G
4	0	1, 3, 5, 7, 9, 11	A	G
5	0	1, 3, 5, 7, 8	A	G
6	1	10	F	B

Table 1. All the DD-paths which make up each level-i path for module FIND.

6.1 Stability Measures for Software Maintenance

The stability of a program is the resistance of the program to the potential ripple effect caused by program changes. Before considering the stability of a program, it is necessary to develop a measure for the stability of a module. The stability of a module is a measure of the resistance to the potential ripple effect caused by modifications to the module on other modules in the program. There are two aspects of the stability of a module: the logical aspect and the performance aspect. The logical stability of a module is a measure of the resistance to the ripple effect of such a modification on other modules in the program in terms of logical considerations. The performance stability of a module is a measure of the resistance to the ripple effect of such a modification on other modules in the program in terms of performance considerations. We have developed logical stability measures for a program and the modules of which the program is composed. Performance stability measures will be developed in the future. The stability measures are being developed with the following requirements to increase their applicability and acceptance:

1) Ability to validate the measures,
2) Consistency with current design methodologies,
3) Utilization in comparing different designs, and
4) Diagnostic ability.

It should be noted that the stability measures being described are not in themselves indicators of program maintainability, rather a significant factor contributing to program maintainability. Although the measures being described estimate program stability, they must be utilized in conjunction with the other attributes affecting program maintainability.

Our preliminary results on the development of logical stability measures have been presented in [8]. However, much work remains to be done in the application of these stability measures to the design phase, the development of performance stability measures, and the development of automated restructuring techniques based upon these measures. The stability measures must also be validated.

6.2 Measures for Program Understandability

In another area of software quality, we have established a foundation for

examining software understandability and have developed a method for estimating it. Our approach to this subject is guided by two special considerations. First, we are primarily concerned with the maintenance phase of a software system. Therefore, we deal with existing software systems and not systems still in the design or coding phases. Secondly, we consider not only the program source code but also other software system documents, such as flowcharts, block diagrams, design documents, separate program documentation, and any other types of documentation which may exist. The reason is that when someone is attempting to understand a program, he uses as much information on the software system as possible. Thus, each component in the document contributes its share to the overall understandability of the system.

Our measure of understandability for a software system is based on Shannon's information theory [58] and to a lesser degree on Halstead's software science theory [59]. We hypothesize that the more information that is contained in a software system, or a part of it, the more time that is required to understand the system, or that part of it. Furthermore, we contend that the more time that is necessary to understand the system, the less is its understandability. In assessing the information content of a "message", Shannon's theory says that the amount of information in a message of a specific length is based on the total number of messages possible of that length. Relating this theory to our approach, we define an understandability measure called the description volume which is based on the "information content" of each component of the software system document, such as program code, flowcharts, block diagrams, etc. Characterizing each component as consisting of basic elements and relationships among these elements, we compute a description volume based on the number of possible configurations of the particular component given the number and type of elements and their relationships. The more configurations that are possible, the more information that is contained in the particular configuration that exists.

Our method relates to Halstead's software science in the sense that one component in the computation of the description volume for program code is the program volume [59]. We then extend these ideas by computing additional components for the description volume which are based on the control structure of the program and on the relationships which exist among the data structures.

Since there can be many different types of components in the documentation of a software system, our method provides for the computation of a description volume for each component. Each component is assessed based on a set of characteristic measures selected for that component. From these characteristic measures, the description volume is computed. The sum of the description volumes of all components then yields the understandability of the software system.

At present, our approach is expected to work well when the documentation of the program under consideration is complete and well organized. Substantial further work is required to generalize our results in order to make them practical.

7.0 DYNAMIC MONITORING OF SOFTWARE BEHAVIOR

In this area, we have investigated dynamic monitoring of software behavior for maintenance purpose. A software system that can monitor its own behavior during its execution (dynamic monitoring) is called self-metric software. Assertions are usually used to achieve dynamic monitoring. They can provide documentation that helps the maintenance programmer avoid modification errors, and assist him in the detection of any errors that are introduced during modification.

Our results, which have been presented in [60-62], expanded existing assertion concepts to include array data structures which have been virtually ignored in all previous work. Initially, we developed a record oriented approach toward array data structures. Based on this approach, we have presented an assertion technique that will enable effective monitoring of most linear list data structures. We believe this is a significant step in the development of dynamic monitoring as an effective software development tool. We also discussed some important issues relative to the implementation of the assertion concepts and use JOVIAL as an example. We have demonstrated our assertion techniques by incorporating them into JAVS [60], and given some figures concerning the overhead involved [62]. More research is needed to study the use and benefits of dynamic monitoring throughout the whole software life cycle with respect to improving software reliability [63] and maintainability. We need to determine how easily assumptions and decisions can be defined and explicitly stated during software design, and how many of these assertions

can be effectively written and used. Although assertions appear to be a promising tool for detecting errors introduced during software maintenance, we need to determine how effective this type of dynamic monitoring will be in detecting the errors which may be encountered during the maintenance phase.

8.0 REFERENCES

[1] Yau, S. S., Collofello, J. S. and MacGregor, T., "Ripple Effect Analysis of Software Maintenance," Proc. COMPSAC 78, November 1978, pp. 60-65.

[2] Rye, P., Bamberger, F., Ostanek, W., Brodeur, N. and Goode, J., Software Systems Development: A CSDL Project History, RADC-TR-77-213, pp. 33-41. A-42186.

[3] Goodenough, J. B. and Zara, R. V., "The Effect of Software Structure on Software Reliability, Modifiability, and Reusability: A Case Study and Analysis," Softech Inc., July 1974, p. 82.

[4] McCall, J. A., Richards, P. K. and Walters, G. F., Factors in Software Quality, Volumes I, II, and III, General Electric Company, pp. 2-3, 3-5, 7-9.

[5] Goullon, H., Isle, R. and Lohr, K., "Dynamic Restructuring in an Experimental Operating System," Proc. Third Intl. Conf. on Software Engineering, 1978, pp. 295-304.

[6] Ringland, G. and Trice, A. R., "Pilot Implementations of Reliable Systems," Software Practice and Experience, Vol. 8, May-June 1978, pp. 323-339.

[7] Yourdon, E. and Constantine, L., Structured Design, Yourdon, Inc., 1976, p. 392.

[8] Yau, S. S. and Collofello, J. S., "Some Stability Measures for Software Maintenance," Proc. COMPSAC 79, November 1979, pp. 674-679.

[9] Yau, S. S. and Collofello, J. S. and Hsieh, C. C., Ripple Effect Analysis for Large-Scale Software Maintenance--A Handbook, Vol II and III of Final Technical Report.

[10] Yau, S. S. and Collofello, J. S., Performance Considerations in the Maintenance Phase of Large-Scale Systems, RADC-TR-79-129, June 1979. A072380.

[11] Yau, S.S. and Collofello, J.S., Performance Ripple Effect Analysis for Large-Scale Software Maintenance, RADC-TR-80-55, December 1979.

[12] Allen, F. E., "Interprocedural Data Flow Analysis," Information Processing 74, North-Holland Pub. Co., Amsterdam, 1974, pp. 398-402.

[13] Paige, M. R., "Program Graphs, an Algebra, and Their Implications for Programming," IEEE Trans. on Software Engineering, Vol. SE-1, No. 3, September 1975, pp. 286-291.

[14] Paige, M. R., "On Partitioning Program Graphs," IEEE Trans. on Software Engineering, Vol. SE-3, No. 6, November 1977, pp. 386-393.

[15] Pratt, T. W., "A Hierarchical Graph Model of the Semantics of Programs," Proc. 1969 Spring Joint Computer Conf., Vol. 34, May 1969, pp. 813-825.

[16] Pratt, T. W., "Pair Grammars, Graph Languages, and String-to-Graph Translations," Journal of Computer and System Sciences, No. 5, 1971, pp. 560-595.

[17] Pratt, T. W., A Theory of Programming Languages, Part I, University of Texas at Austin, UTEX-CCSN-41, July 1975.

[18] Kunii, T. L., Harada, M. and Saito, M., "RGT: The Recursive Graph Theory as a Theoretical Basis of a System Design Tool 'Design Tool'--With an Application to Medical Information System Design," Proc. of MEDIS 78, pp. 503-507.

[19] Kunii, T. L. and Harada, M., A Design Process Formalization, Dept. of Information Science, University of Tokyo, Tokyo, Japan, 1979.

[20] Kunii, T. L. and Buchmann, A. P., Evolutionary Drawing Formalization in an Engineering Database Environment, Dept. of Information Science, University of Tokyo, Tokyo, Japan, 1979.

[21] Schaefer, M., A Mathematical Theory of Global Program Optimization, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973.

[22] Peterson, W. W., Kasami, T. and Tokura, N., "On the Capabilities of While, Repeat, and Exit Statements," Comm. ACM, Vol. 16, No. 8, August 1973, pp. 503-512.

[23] Melton, R. A., "Automatically Translating FORTRAN to IFTRAN," Computer Science and Statistics: 8th Annual Symposium on the Interface, pp. 291-296.

[24] Pazel, D. P., "Mathematical Construct for Program Reorganization," IBM Journal of Research and Development, November 1975, pp. 575-581.

[25] Hecht, M. S., Flow Analysis of Computer Programs, Elsevier North-Holland 1977.

[26] Sholl, H. A. and Booth, T. L., "Software Performance Modeling Using Computation Structures," IEEE Trans. on Software Engineering, Vol. SE-1, No. 4, December, 1975, pp. 414-420.

[27] Kodres, U. R., "Analysis of Real-Time Systems by Data Flowgraphs," IEEE Trans. on Software Engineering, Vo. SE-4, No. 3, May 1978.

[28] Fosdick, L. D. and Osterweil, L. J., "Data Flow Analysis in Software Reliability," Computing Surveys, Vol. 8, No. 3, September 1976, pp. 305-330.

[29] Fosdick, L. D. and Osterweil, L. J., "The Detection of Anomalous Interprocedural Data Flow," Proc. Second Intl. Conf. on Software Engineering, October 1976, pp. 624-628.

[30] Brown, J. R. and Nelson, E. C., Functional Programming, RADC Technical Report No. 78-24, February 1978.

[31] Goguen, J. A., Thatcher, J. W., et al., "Abstract Data Types as Initial Algebras and the Correctness of Data Representations," Proc. of Conf. on Computer Graphics, Pattern Recognition and Data Structures, 1975, pp. 89-93.

[32] Guttag, J., "Abstract Data Types and the Development of Data Structures", Comm. ACM, Vol. 20, No. 6, June 1977, pp. 396-404.

[33] Guttag, J. and Horning, J. J., "The Algebraic Specification of Abstract Data Types," Acta Informatica, Vol. 10, No. 1, 1978, pp. 27-52.

[34] Guttag, J., "Notes on Type Abstraction," Proc. Specifications of Reliable Software, 1979, pp. 36-46.

[35] Earley, J., "Toward an Understanding of Data Structures," Comm. ACM, Vol. 14, No. 10, October 1971, pp. 617-627.

[36] Honig, W. L. and Carlson, C. R., "Toward an Understanding of (Actual) Data Structures," Computer Journal, Vol. 21, No. 2, May 1978, pp. 98-104.

[37] Jones, C. B., "Constructing a Theory of a Data Structure as an Aid to Program Development," Acta Informatica, Vol. 11, 1979, pp. 119-137.

[38] Hamilton, M. and Zeldin, S., "Higher Order Software--A Methodology for Defining Software," IEEE Trans. on Software Engineering, Vol. SE-2, No. 1, March 1976, pp. 9-32.

[39] Bell, T. E., Bixler, D. C., and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 49-59.

[40] Alford, M. W., "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 60-69.

[41] Alford, M. W., "R-nets: A Graph Model for Real-time Software Requirements," Proc. Symp. on Computer Software Engineering, MRI Symposium Series, Vol. XXIV, Brooklyn, N.Y., Polytechnic Press of the Polytechnic Institute of New York, 1976.

[42] Ross, D. T., "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 16-34.

[43] Jackson, M. A., "Information Systems: Modelling, Sequencing and Transformations," Proc. 3rd Intl. Conf. on Software Engg., May 1978, pp. 72-81.

[44] Ruggiero, W., Estrin, G., Fenchel, R., Razouk, R., Schwabe, D., and Vernon, M., "Ananlysis of Data Flow Models Using the SARA Graph Model of Behavior," Proc. National Computer Conf., Vol. 48, 1979, pp. 975-988.

[45] Teichroew, D. L. and Hershey, E. A., III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 41-48.

[46] Ramamoorthy, C. V. and So, H. H., "Software Requirements and Specifications: Status and Perspectives," IEEE Tutorial: Software Methodology, 1978, pp. 43-164.

[47] Codd, E. F., "A Relational Model for Large Shared Data Banks," Comm. ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

[48] Fischer, K. F., "A Test Case Selection Method for the Validation of Software Maintenance Modifications," Proc. COMPSAC 77, November 1977, pp. 421-426.

[49] Hamlet, R., "Testing Programs with the Aid of a Compiler," IEEE Trans. on Software Engineering, Vol. SE-3, No. 4, July 1977, pp. 297-290.

[50] Panzl, D., "Automatic Revision of Formal Test Procedures," Proc. Third Intl. Conf. on Software Engineering, May 1978, pp. 320-326.

[51] Howden, W. E., "Reliability of the Path Analysis Testing Strategy," IEEE Trans. on Software Engineering, September 1976.

[52] Miller, E. F., Paige, R., Benson, P. and Wisehart, W. R., "Structural Techniques of Program Validation," Digest of Papers COMPCON 74, February 1974, pp. 161-164.

[53] Huang, J. C., "An Approach to Program Testing," ACM Computing Surveys, Vol. 7, No. 3, September 1975, pp. 113-128.

[54] Clarke, L. A., "A System to Generate Test Data and Symbolically Execute Programs," IEEE Trans. on Software Engineering, Vol. SE-2, No. 3, September 1976, pp. 215-222.

[55] Ramamoorthy, C. V., Ho, S. F. and Chen, W. T., "On the Automated Generation of Program Test Data," IEEE Trans. on Software Engineering, Vol. SE-2, No. 4, December 1976, pp. 293-300.

[56] Howden, W. E., "Methodology for the Generation of Program Test Data," IEEE Trans. on Computers, Vol. C-24, No. 5, May 1975, pp. 554-560.

[57] Miller, E. F., Jr., Methodology for Comprehensive Software Testing, RADC Technical Report No. 75-161, June 1975, 150 pages, A013111.

[58] Aczel, J. and Daroczy, Z., On Measures of Information and Their Characterizations, Academic Press, Inc., New York, 1975.

[59] Halstead, M. H., Elements of Software Science, Elsevier, New York, 1977.

[60] Yau, S. S. and Ramey, J. L., Dynamic Monitoring for Linear List Data Structures, RADC Technical Report No. 79-128, June 1979.

[61] Yau, S. S. and Ramey, J. L., "Assertion Techniques for Dynamic Monitoring of Linear List Data Structures," Proc. COMPSAC 79, November 1979, pp. 606-611.

[62] Yau, S. S., Ramey, J. L., and Nicholls, R. A., "Assertion Techniques for Dynamic Monitoring of Linear List Data Structures," to be published in Journal of Systems and Software. (This paper contains additional results, including the discussion on usefulness of dynamic monitoring, hierarchical checking, and overhead statistics on using the assertion techniques).

[63] S. S. Yau and MacGregor, T. E., On Software Reliability Modeling, RADC Technical Report No. 79-127, June 1979, 49 pages, A072420.

9.0 PUBLICATIONS AND PRESENTATIONS

Besides the results of the research presented in this report, some have already been published or presented both in preliminary and complete forms. The publications and presentations are grouped in the following categories: (1) papers, (2) technical reports, and (3) presentations related to the project.

9.1 Papers

1. S. S. Yau, J. S. Collofello and T. M. Macgregor, "Ripple Effect Analysis for Software Maintenance," Proc. Second International Conf. on Computer Software and Applications, (COMPSAC 78), November 14-16, 1978, pp. 60-65.

2. S. S. Yau and J. L. Ramey, "Assertion Techniques for Dynamic Monitoring of Linear List Data Structures," Proc. Third International Conf. on Computer Software and Applications, (COMPSAC 79), November 5-8, 1979, pp. 606-611.

3. S. S. Yau and J. S. Collefello, "Some Stability Measures for Software Maintenance," Proc. Third International Conf. on Computer Software and Applications (COMPSAC 79), November 5-8, 1979, pp. 674-679.

4. S. S. Yau, J. L. Ramey and R. A. Nicholls, "Assertion Techniques for Dynamic Monitoring of Linear List Data Structures," to be published in Journal of Systems and Software.

9.2 Technical Reports

1. S. S. Yau and T. E. MacGregor, "On Software Reliability Modeling," Interim Technical Report RADC-TR-79-127, June, 1979, 49 pages.

2. S. S. Yau and J. L. Ramey, "Dynamic Monitoring for Linear List Data Structures," Interim Technical Report RADC-TR-79-128, June 1979, 111 pages, A-72381.

3. S. S. Yau and J. S. Collofello, "Performance Considerations in the Maintenance Phase of Large-Scale Software Systems," Interim Technical Report RADC-TR-79-129, June 1979, 44 pages.

4. S. S. Yau and J. S. Collofello, "Performance Ripple Effect Analysis for Large-Scale Software Maintenance," Interim Technical Report RADC-TR-80-55, December 1979.

9.3 Presentations

1.† S. S. Yau, "Recent Advances in Computer Systems Reliability," Purdue University, joint seminar of Departments of Computer Science and Electrical Engineering, Lafayette, Indiana, February 8, 1977.

2. S. S. Yau, "Design of Self-Metric Software," Seminar, Rome Air Development Center, Rome, New York, March 17, 1977.

3.† S. S. Yau, "Design of Self-Checking Software," University of Pittsburgh, Computer Science Department, Pittsburgh, PA, March 23, 1977.

4.† S. S. Yau, "Some Techniques for Improving Computer Systems Reliability and Maintainability," 1977 U.S.S.R. Popov Society Congress, Moscow, U.S.S.R., May 17-19, 1977.

5.† S. S. Yau, "Some Techniques for Improving Reliability and Maintainability of Computing Systems," Seminar, Bioengineering/Advanced Automation Program, University of Missouri-Columbia, December 13, 1977.

6.† S. S. Yau, "Recent Advances in Reliability of Computer Systems," Talk, American Society for Quality Control Chicago Section Meeting, Chicago, February 8, 1978.

---

†These presentations were made at no cost to the contract.

7.† S. S. Yau, "Some Techniques for Improving Reliability and Maintainability of Computing Systems," Joint Seminar, Department of Computer and Information Science, Ohio State University and IEEE Computer Society Columbus, Ohio, February 16, 1978.

8. S. S. Yau, "On Software Maintenance," Seminar, Rome Air Development Center, Rome, New York, May 4, 1978.

9. S. S. Yau, "Effect of Program Modification on Software Performance," Minnowbrook Workshop on Software Performance Modelling and Error Analysis, Blue Mountain Lake, New York, September 19-21, 1978.

10.† S. S. Yau, "Some Techniques for Improving Reliability and Maintainability of Computing Systems," Seminar, University of Illinois at Urbana, Department of Electrical Engineering, September 27, 1978.

11. S. S. Yau, "Ripple Effect Analysis for Software Maintenance," Second International Computer Software and Applications Conference (COMPSAC 78), Chicago, November 14-16, 1979 (see Section 3.1).

12.† S. S. Yau, "On Effective Software Maintenance Methodology," Seminar, University of California at Berkeley, Department of Electrical Engineering and Computer Science, November 30, 1979.

13.† S. S. Yau, "Design of Reliable Software," Seminar, Nippon Electric, Ltd., Tokyo, Japan, April 10, 1979.

14.† S. S. Yau, "Software Product Engineering," Seminar, sponsored by Sundai Research and Development, April 11-13, Tokyo, Japan.

15.† S. S. Yau, "Design of Reliable Software," Seminar, NEC-Toshiba Information System, Inc., Tokyo, Japan, April 16, 1979.

16. S. S. Yau, "A Technique for Ripple Effect Analysis for Software Maintenance," Seminar, Rome Air Development Center, Rome, New York, May 10, 1979.

17. S. S. Yau and J. S. Collofello, "Some Measures on Software Maintainability," Second Minnowbrook Workshop on Performance Evaluation, Blue Mountain Lake, New York, July 31-August 3, 1979.

18.† S. S. Yau, "Some Techniques for Improving Computer Reliability and Maintainability," Seminar, Chinese Institute of Electronics Meeting, Hangzhow, China, September 20, 1979.

19. S. S. Yau, "Assertion Techniques for Dynamic Monitoring of Linear List Data Structures," COMPSAC 79, Chicago, Illinois, November 5-8, 1979.

20. S. S. Yau, "Some Stability Measures for Software Maintenance," COMPSAC 79, Chicago, Illinois, November 5-8, 1979.

21.† S. S. Yau, "Development of Reliable Large-Scale Software Systems," Seminar, IEEE Taiwan Section Meeting, Taipei, Taiwan, December 27, 1979.

10.0 TECHNICAL PERSONNEL

During the period of this study, the following Northwestern University faculty and graduate students contributed to the research effort of this contract:

	1976	1977	1978	1979	1980
Principal Investigator and Project Director	Starting Aug. 1				Ending Jan. 15
Stephen S. Yau	X	X	X	X	X
Faculty Investigators					
Roger C. Cheung	X	X	--	--	--
J. S. Collofello*	--	--	--	X	X
R. Konakovsky**	--	--	X	X	--
U. Gupta	--	--	--	X	--
Graduate Students					
M. Y. Chern	X	X	--	--	--
T. MacGregor	X	X	X	--	--
P. C. Tam	X	X	--	--	--
J. S. Collofello*	--	X	X	--	--
J. Ramey	--	X	X	--	--
Z. Kishimoto	--	X	X	X	X
C. C. Hsieh	--	--	X	X	X
L. E. Rich	--	--	X	X	--
P. Grabow	--	--	X	X	--
P. Ham	--	--	--	X	X
R. Nicholl	--	--	--	X	X
C. Chieng	--	--	--	X	X
C. K. Chang	--	--	--	X	X

In addition, Professor C. V. Ramamoorthy of the University of California at Berkeley served as a consultant to this contract in 1978 and 1979, and Professor T. L. Kunii of the University of Tokyo served as a consultant to this contract in 1979.

---

*Dr. Collofello continued to contribute to the research effort after he completed his Ph.D. degree as a visiting assistant professor and consulting after August, 1979.

**Visiting assistant professor.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


DAT
FILM